Hardware/Software Co-Design of Heterogeneous Manycore Architectures

Süleyman Savas

Supervisors:
Tomas Nordström
Zain Ul-Abdin
Abstract

In the era of big data, advanced sensing, and artificial intelligence, the required computation power is provided mostly by multicore and manycore architectures. However, the performance demand keeps growing. Thus the computer architectures need to continue evolving and provide higher performance. The applications, which are executed on the manycore architectures, are divided into several tasks to be mapped on separate cores and executed in parallel. Usually these tasks are not identical and may be executed more efficiently on different types of cores within a heterogeneous architecture. Therefore, we believe that the heterogeneous manycores are the next step for the computer architectures. However, there is a lack of knowledge on what form of heterogeneity is the best match for a given application or application domain. This knowledge can be acquired through designing these architectures and testing different design configurations. However, designing these architectures is a great challenge. Therefore, there is a need for an automated design method to facilitate the architecture design and design space exploration to gather knowledge on architectures with different configurations. Additionally, it is already difficult to program manycore architectures efficiently and this difficulty will only increase further with the introduction of heterogeneity due to the increase in the complexity of the architectures, unless this complexity is somehow hidden. There is a need for software development tools to facilitate the software development for these architectures and enable portability of the same software across different manycore platforms.

In this thesis, we first address the challenges of the software development for manycore architectures. We evaluate a dataflow language (CAL) and a source-to-source compilation framework (Cal2Many) with several case studies in order to reveal their impact on productivity and performance of the software. The language supports task level parallelism by adopting actor model and the framework takes CAL code and generates implementations in the native language of several different architectures.

In order to address the challenge of custom hardware development, we first evaluate a commercial manycore architecture namely Epiphany and identify its demerits. Then we study manycore architectures in order to reveal possible uses
of heterogeneity in manycores and facilitate choice of architecture for software and hardware development. We define a taxonomy for manycore architectures that is based on the levels of heterogeneity they contain and discuss the benefits and drawbacks of these levels. We finally develop and evaluate a design method to design heterogeneous manycore architectures customized based on application requirements. The architectures designed with this method consist of cores with application specific accelerators. The majority of the design method is automated with software tools, which support different design configurations in order to increase the productivity of the hardware developer and enable design space exploration.

Our results show that the dataflow language, together with the software development tool, decreases software development efforts significantly (25-50%), while having a small impact (2-17%) on the performance. The evaluation of the design method reveal that the performance of automatically generated accelerators is between 96-100% of the performance of their manually developed counterparts. Additionally, it is possible to increase the performance of the architectures by increasing the number of cores and using application specific accelerators, usually with a cost on the area usage. However, under certain circumstances, using accelerator may lead to avoiding usage of large general purpose components such as the floating-point unit and therefore improves the area utilization. Eventually, the final impact on the performance and area usage depends on the configurations. When compared to the Epiphany architecture, which is a commercial homogeneous manycore, the generated manycores show competitive results. We can conclude that the automated design method simplifies heterogeneous manycore architecture design and facilitates design space exploration with the use of configurable parameters.
I would like to express my appreciation to my supervisors Tomas Nordström and Zain-Ul-Abdin for their guidance. I would also like to thank to my support committee members Bertil Svensson, Thorstein Rögnvaldsson, and Anders Åhlander for their valuable comments on my research. I want to express my gratitude to Stefan Byttner and Håkan Petterson for their support as the study coordinators and to Eva Nestius for the help with all kinds of issues.

My colleagues Mahsa, Nesma, Essayas, Sebastian, Erik and our project partner Madhura Purnaprajna deserve special thanks for all the discussions and suggestions. I would like to thank to all my other colleagues and Halmstad Research Students’ Society (HRSS) members.
Contents

1 Introduction 1
  1.1 Motivation ..................................................... 1
  1.2 Research Questions ........................................... 5
  1.3 Approach ...................................................... 5
  1.4 Contributions ................................................ 7

2 Background 11
  2.1 Manycore Architectures ....................................... 11
  2.2 Software Development for Manycores .......................... 14
  2.3 Dataflow Programming Model .................................. 15
  2.4 CAL Actor Language ............................................ 16
  2.5 Streaming Applications / Dataflow Programs ................. 18
    2.5.1 Two Dimensional Inverse Discrete Cosine Transform (2D-IDCT) .............................................. 19
    2.5.2 QR Decomposition ........................................... 20
    2.5.3 Synthetic Aperture Radar Systems and Autofocus Criterion Calculation ............................................ 20
  2.6 Evaluation of Software/Hardware Development Tools and Manycore Architectures ............................... 21
  2.7 Design of Manycore Architectures .............................. 21
    2.7.1 RISC-V and Rocket Chip .................................. 22
    2.7.2 Rocket core and RoCC Interface ........................... 22
    2.7.3 Chisel ...................................................... 23
    2.7.4 Cal2Many ................................................... 25
    2.7.5 TURNUS .................................................... 27

3 Heterogeneity in Manycore Architectures 29
  3.1 Levels of Heterogeneity ....................................... 30
  3.2 Conclusion .................................................... 32
## 4 Domain-Specific Heterogeneous Manycore Design

4.1 Application Development ........................................ 38  
4.2 Analysis and Code Generation ................................. 38  
4.3 Accelerator Integration ........................................ 43  
4.4 System Integration ............................................. 45

## 5 Summary of the Papers

5.1 Evaluation of Software Development Tools for Manycores - Paper I ........................................ 49  
5.2 Parallel QR Decomposition on Epiphany Architecture - Paper II ....................................... 49  
5.3 Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis- Paper III ........................................ 50  
5.4 Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs - Paper IV ........................................ 51  
5.5 Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit - Paper V ........................................ 51  
5.6 A Configurable Two Dimensional Mesh Network-on-Chip Implementation in Chisel - Paper VI ........................................ 52  
5.7 A Framework to Generate Domain-Specific Manycore Architectures from Dataflow Programs - Paper VII ........................................ 53

## 6 Conclusions and Future Work

6.1 Conclusions .................................................. 55  
6.2 Future Work .................................................. 57

References ......................................................... 59

Paper I ......................................................... 69  
Paper II ......................................................... 81  
Paper III ....................................................... 89  
Paper IV ........................................................ 97  
Paper V .......................................................... 127  
Paper VI .......................................................... 135  
Paper VII .......................................................... 145
# List of Figures

1.1 The microprocessor trend data for the years between 1971 and 2017. ............................... 2

2.1 Overview of Epiphany-V manycore architecture from Adapteva [70]. ............................. 12

2.2 Overview of Myriad2 architecture from Movidius [66]. .................. 13

2.3 A sample streaming application - the IDCT2D block of MPEG-4 Simple Profile decoder; represented as interconnected actors (the boxes are the actors and the triangles are input/output ports, which can be connected to any source or sink). [79] ............................. 16

2.4 A simple splitting actor in CAL actor language. .............................. 17

2.5 A simple combining actor in CAL actor language. .......................... 17

2.6 A network of two actors in CAL actor language. .......................... 18

2.7 Connections between the core and the accelerator provided by the RoCC interface. Request and response connections are separated for a clearer illustration. ............................. 23

2.8 2to1 multiplexer hardware written in Chisel. ............................. 24

2.9 An overview of the Cal2Many framework. ................................. 26

4.1 A generic view of the design flow for developing domain-specific heterogeneous manycore architectures. ................................. 37

4.2 Realization of the design flow for developing domain-specific heterogeneous manycore architectures. ................................. 39

4.3 An action of QR decomposition application written in CAL actor language. ................................. 40

4.4 Bottleneck analysis results of Autofocus criterion calculation application implemented in CAL, provided by TURNUS. .................. 41

4.5 Bit fields of the custom instruction of RISC-V instruction set. ................................. 42

4.6 A crossbar network with 8 cores and 16 switches on the left and a two dimensional mesh network with 16 cores and 16 routers on the right. ................................. 45
List of Tables

3.1 Main levels of heterogeneity and example commercial architectures. ........................................... 32
List of Publications

This thesis summarizes the following publications.


**Contribution:** I implemented 2D-IDCT in CAL and C languages targeting Epiphany architecture. The second author provided generated code from CAL implementation. I have executed the generated and the hand-written implementations on the architecture and collected the results. I led the discussion and paper writing.


**Contribution:** I implemented different versions of one of the algorithms for QRD in CAL and C languages. CAL code is converted into C by the third author. I executed the hand-written and generated implementations on Epiphany architecture and collected results. The discussions are led by the first three authors and the paper writing is led by me.

**Contribution:** I developed the division hardware, verified its functionality, collected the results, and led the discussions and paper writing.


**Contribution:** I developed the design method, the code generation tool, and the hardware library, implemented the case studies, collected the results, and led the discussion and paper writing.

**Paper V** Süleyman Savas, Yassin Atwa, Zain Ul-Abdin, and Tomas Nordström. "Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit", accepted to International Symposium on VLSI Design (ISVLSI), Miami, Florida, USA, July 15-17, 2019.

**Contribution:** I developed the square root hardware with the help from second author, verified its functionality, collected the results, and led the discussions and paper writing.

**Paper VI** Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "A Configurable Two Dimensional Mesh Network-on-Chip Implementation in Chisel", submitted to 32nd IEEE International System-on-Chip Conference (SOCC), Sept. 3-6, 2019, Singapore.

**Contribution:** I developed the two dimensional mesh network-on-chip router in Chisel in collaboration with a project partner. The network interface is developed by me. All network implementations, result collection, analysis of the results and paper writing is performed by me. The other authors contributed with discussion on the requirements of the NoC and feedback about the text and content.


**Contribution:** I integrated the network-on-chip router that is developed in paper VI to the rocket chip generator, extended the code generation tool and the hardware library, implemented the case studies, collected results, and led the discussion and paper writing.
Other Publications


Chapter 1
Introduction

1.1 Motivation

Processor technology has evolved incredibly in the last 70 years. The increased power of computation has continuously enabled new application areas and these new applications have craved for more computation power. This process is continuing even today, however, with a smaller pace on the hardware side. The application areas that are available today such as autonomous vehicles, smart cities, climate and environment monitoring, and scientific computations consist of many applications, which require very powerful computers due to dealing with large amount of data and performing extensive amounts of computations. Majority of the applications collect data through different types of scattered sensors and move some of the computations near to these sensors to address the concerns of response time requirement, bandwidth cost saving, as well as data safety and privacy [81]. Performing the heavy computations near the edge sensors requires high-performance embedded systems. The performance demands continue growing as the size of the data and the capabilities of the applications increase. In order to satisfy the growing performance requirements, enable new application areas and increase the quality and capability of applications, the processors need to continue improving.

The two major forces behind the progress of processors are the improvements in technology and the improvements in computer architecture design. Even though the latter one has been less consistent, the prior one has been quite steady [41]. In 1965, Gordon Moore estimated that the number of transistors per square inch on integrated circuits double every year [64]. Later, in 1975, he changed the time span to 24 months [65]. The increase in the number of transistors was enabled by decreasing the dimensions of the transistors. Smaller sized transistors enabled higher clock frequencies as well. However, the power consumption soon turned out to be a bottleneck for this advancement.

The power consumption depends on capacitance of the transistor, voltage level and clock frequency [90] as given in equation 1.1. The increase in clock fre-
CHAPTER 1. INTRODUCTION

quency or in transistor density could be compensated by decreasing the voltage level. The relation between the transistor density and the power consumption is described by Dennard scaling [23]. When the transistors get smaller, the voltage and current should scale down along with the power consumption. This reduction compensates the increase in power consumption caused by the growth in the number of the transistors. Therefore, despite increasing the number of the transistors, the power consumption for a given area of silicon stays constant due to the smaller dimensions of the transistors. However, Dennard scaling has come to an end around 2004 due to the limitations on current and voltage scaling [14, 41]. If the supply voltage continues decreasing, circuit delay increases and leakage energy dominates any reduction in switching energy [25, 52]. Since the voltage could not be decreased anymore, the increases in clock frequency could not be compensated. Therefore the increase in clock frequency has almost come to a stop, after the scaling has ended. This can be seen in Figure 1.1. In addition to the clock frequency, the figure presents the trends for transistor count, performance, power consumption, and number of cores of microprocessors between 1971 and 2017.

\[ P = CV^2 f \] (1.1)

**Figure 1.1:** The microprocessor trend data for the years between 1971 and 2017.
1.1. MOTIVATION

By the end of Dennard Scaling, the changes in voltage and current scaling forced the microprocessor industry to utilize multiple efficient small cores instead of an inefficient, fat single core. Thus, the transition from single core to multiple cores began. The first general-purpose processor to have multiple processing cores on the same die was the POWER4 [86] by IBM, which was released in 2001. However, the processors with multiple cores became a trend around 2005.

The first multicore processors were developed by creating identical copies of a general purpose core. These architectures are homogeneous in terms of processing cores. In such architectures, the performance and power consumption characteristics of cores are identical. Later, in order to increase the performance and power efficiency of processors, several academic and commercial heterogeneous architectures consisting of different types of cores on the same chip have been developed [55, 54, 51, 72, 16, 4, 61, 66, 77]. It was shown that, given the same die size, heterogeneous architectures are more efficient than the homogeneous counterparts in terms of power and performance [55, 15, 60, 63]. It is therefore quite clear that the transition to heterogeneous architectures is the next step of increasing the efficiency of the processors.

We define heterogeneity as combination of two or more cores with different implementations or different complete instruction sets, on the same chip. To be able to understand pros and cons of the heterogeneity, one needs to understand the different forms of it. For instance, ARM’s big.Little architecture [4] has a mild form of heterogeneity where two different implementations of the same instruction set architecture (ISA) is combined on the same chip. IBM’s Cell processor [51] presents a slightly more complicated form of heterogeneity by combining cores with two different ISAs. Movidius’ Myriad 2 [66] increases the complexity a bit further by adding hardware accelerators on top of the heterogeneity form that is introduced by the Cell processor. Different forms of heterogeneity have different advantages and disadvantages. The details of these heterogeneity forms are given in Chapter 3.

The complexity of the architectures increases with the number of the cores and with the introduction of the heterogeneity. As the degree of heterogeneity increases, the architecture becomes more complex due to having different types of processing cores to deal with. This complexity has an impact on both software and hardware development. We will first go through the software problems and then the hardware problems.

In order to exploit the parallelism in the multicore/manycore architectures, the software applications with sequential legacy code must be re-implemented. Hennessy and Patterson [42] suggest to use domain-specific languages with support for parallelism to program manycore architectures more efficiently. In addition to agreeing to this statement, we believe that programming models with parallelism support such as the dataflow programming model [67] can facilitate the application development for manycore architectures. However, to be able to develop efficient parallel applications, new models and languages
alone are not enough. The developer needs to have a deep understanding on the capabilities of the target architecture as well to make best use of the resources and achieve the desired efficiency.

Manycore architectures introduce further challenges and requirements. If the cores share memory, memory coherence can be an issue. Additionally, a core-to-core communication mechanism, that uses some form of on-chip communication hardware, is necessary in order to enable the data transfers between the tasks running in parallel. These challenges are common for homogeneous and heterogeneous manycores. However, additional challenges occur when heterogeneity is introduced. For instance, task migration might be more cumbersome if the processing cores are not identical [24]. The core, which is the destination of the migration, might not be supporting some of the operations performed in the task. Or, even worse, it might be executing a different instruction set. In order to support the task migration, the developer might need to write the same task in different languages or compile it with different compilers and store multiple binaries, which increases memory requirements. Even if the task migration is ignored, the developer is required to parallelize the application and deal with mapping the application onto different processing cores with different characteristics.

While developing a homogeneous manycore architecture, the developers can replicate a core instead of re-implementing it many times. Additionally, since the characteristics of each core is identical, their location on the chip does not play much of a role. They can replace each other without affecting anything as they would act identically. However, if the cores differ, then the developers need to implement them individually and their location on the chip might have an impact on the performance of the architecture. For instance, a core with better memory management resources can be placed on the edge of the chip to take care of the off-chip memory accesses. In another scenario, the cores with high computation power can be spread out on the chip to dissipate the heat. There are several design dimensions such as, structure and types of the cores, structure and types of memories, and interconnection network to be considered while designing the manycore architectures. The decision for what configurations to use for the design dimensions can be a challenge itself. All these challenges pose significant obstacles for the manycore architectures and heterogeneity on the path of becoming mainstream.

The design configurations determine the characteristics of the architecture. Many different types of manycore architectures and different forms of heterogeneity can be created by modifying these configurations [63]. The best configurations depend on the requirements of the target application(s). The characteristics of the architecture can be estimated based on the configurations, however, for more accurate results, there is a need to develop the architecture and evaluate certain configurations. Due to the difficulties in developing and manufacturing the processors, researchers have been using simulations to explore the design space of the architectures. There are several software tools
such as Gem5 [13], SimFlex [39], MARSS [73], SimpleScalar [7], BigSim [97], ZSim [78], Graphite [62], Sniper [19], and PriME [30] to simulate architectures. Gem5 and SimFlex are full system simulators, which can simulate entire architectures and provide information on different components. A major problem with the simulators is the simulation time when the system size increases. It can be beyond feasible thresholds or slow down the design space exploration dramatically [96]. Another drawback of the simulators is the lack of hardware synthesis data such as the achievable clock frequencies, resource usage, and area. Therefore, there is a need for a design method and automation tools to design manycore architectures with different design configurations and synthesize them on hardware to run applications faster and analyze their behaviour without relying on the simulators.

We have described the requirements and challenges, which come with the introduction of heterogeneity in manycore architectures, under two categories; software and hardware. We formulate research questions based on these challenges and requirements and then describe the approach that we used to find solutions to these questions.

### 1.2 Research Questions

We divide the challenges of introducing heterogeneity in manycore architectures into two groups; software and hardware.

The research question on the software challenges is as follows:

- What are the benefits and drawbacks of using dataflow programming model, dataflow languages, and software development tools while developing parallel programs?

The research questions covering the hardware challenges are:

- How can we define different levels of heterogeneity and what are the advantages and disadvantages of these levels?

- How to facilitate the design and implementation of domain-specific heterogeneous manycore architectures?

The next section provides the methodology that we have used to answer the research questions.

### 1.3 Approach

We start with describing our approach to address the software challenges and then continue with the approach to address the hardware challenges.

In order to explore the benefits and drawbacks of using software development tools, dataflow programming model and languages, we evaluate the
programming model, a dataflow language and software tools that facilitate development, mapping and execution of parallel programs on different manycore architectures. The evaluation is done in several steps. First, the applications are developed with the dataflow programming language and then with the native programming language of the target architecture. The correctness of the dataflow implementation is verified with a simulator and this implementation is automatically converted into the native language with the software tools. The hand-written and the generated native implementations are executed on the target architecture to collect results for performance and memory usage. These results are analyzed and compared to each other. The productivity results, which are based on the time and effort spent on developing the implementations, are considered for this comparison.

We use CAL actor language [27] as the dataflow language to develop the applications. This language is a high-level domain-specific language with support for task parallelism (in the form of actors) and explicit actor-to-actor communication. We use it to increase the abstraction level for parallel programming and hide the details of the target architecture. In order to generate target specific implementations, we use a software tool called Cal2Many [31] that is a source-to-source compiler developed in-house. It takes CAL code as input and generates compilable code for several different architectures including commercial and academic single core and manycore architectures. We execute the implementations on a parallella board with a 16 core Epiphany chip, which is a commercial homogeneous manycore architecture [71]. Execution results are analyzed with respect to communication and computation to measure the performance of the architecture and the code generation. The footprint and the source line of the codes are used for comparison between the generated implementations and the hand-written native implementations. The drawbacks of the architecture are identified and the software tools are optimized based on the evaluation results. During the execution of the implementations on the parallella board, different mapping strategies are tested. However, there was no difference between the results mostly due to the overhead of the communication mechanism in software. Therefore, those results are not considered further.

In addition to the evaluation of a homogeneous architecture with several applications, we study several other commercial and academic manycore architectures to define heterogeneity and its forms. We classify the studied architectures using these forms and identify their advantages and disadvantages. Later, in order to explore the design space of these architectures we develop a design method to design heterogeneous manycore architectures that fit into one of the forms, where the architectures consist of augmented cores with the same instruction set architecture. The augmentations are instruction extensions, which are in the form of custom hardware accelerators. The accelerators are implemented based on the computational requirements of the applications. Hence, the cores become specialized for the chosen applications. In order to
facilitate the software programming, we continue using CAL actor language along with other software tools to identify the computationally intensive parts of the applications and generate hardware code (accelerators) for these parts. The generated accelerators are integrated to another software tool that can generate architectures consisting of cores, which are augmented with these accelerators. Additionally, we integrate a scalable two dimensional mesh network-on-chip (NoC) to this software tool to realize domain-specific heterogeneous manycore architectures. In order to evaluate the design method, the software tools, and the generated architectures, we develop applications from different domains such as radar signal processing and machine learning, in CAL actor language, push them through the software tools to automatically generate several specialized architectures with different configurations. We measure the performance and the hardware resource usage of these architectures and compare them to some reference architectures including architectures with cores, which have manually developed accelerators. The results help identify the advantages and disadvantages of the architectures and measure the performance of the software tools.

Unfortunately it is a challenge to measure the productivity in a quantitative manner. However, one can imagine the effort of developing a manycore architecture from scratch and generating it automatically with tools by setting a few configuration parameters. The design method still requires some effort to integrate new accelerators to the tools and change configurations based on the requirements of the applications. This effort can be just a few hours of coding and verification for a slightly experienced developer. This is less than the effort required for manually developing the accelerators in our case studies and negligible when compared to developing architectures from scratch.

1.4 Contributions

This thesis addresses challenges in both software and hardware dimensions of parallelism in computer architectures. In terms of software, it aims to evaluate a language and software development tools for decreasing the complexity of parallel programming. In terms of hardware, it targets to increase the efficiency of manycore architectures by introducing heterogeneity and facilitate design space exploration by proposing a design method and automating the main steps of this method with software tools.

The contributions of the thesis can be summarized as below:

- Definition of different heterogeneity levels and a taxonomy to classify manycore architectures based on their heterogeneity level. The aim is to provide guidance to hardware and software developers to choose the best architecture they need by providing advantages and drawbacks of different heterogeneity levels. (Chapter 4)
• Evaluation and optimization of software development tools for manycore architectures through implementation of parallel two dimensional inverse discrete cosine transform (2D-IDCT) on the Epiphany manycore architecture [71]. (Paper I)

• Evaluation of the CAL actor language, software development tools and the Epiphany architecture through implementation and evaluation of different versions of parallel QR decomposition [33] (Paper II)

• Implementation and evaluation of custom hardware for single-precision floating-point division based on a novel method [43, 44] on an FPGA. (Paper III)

• A methodology to design domain-specific heterogeneous manycore architectures with custom-designed tiles. Generation of single tiled architectures and evaluation of this method with two case studies. (Paper IV)

• Automation of the design method with software tools including development of a code generation back-end that performs hardware/software co-design by generating C and Chisel code from CAL applications. The other steps of the design method are automated with other tools including TURNUS [20] and rocket chip generator [5]. (Paper IV)

• Implementation and evaluation of custom hardware for single-precision floating-point square root based on a novel method [43, 44] on an FPGA. (Paper V)

• Development of a two dimensional mesh network-on-chip router and a network interface in Chisel. The interface is for the RISC-V (rocket) core that is generated by the rocket chip generator. Network implementations with different sizes are analyzed in terms of hardware resource usage, clock rate, latency and throughput. (Paper VI)

• Extension of the code generation tool that is described in paper IV to support core-to-core communication in software and generation of code for multiple cores with multiple accelerators. Extension of the rocket chip generator with the scalable two dimensional mesh network-on-chip, that is presented in paper VI, to support generation of architectures with many cores. Generation of several architectures with different number of cores and configurations for two case studies from different domains.
1.4. CONTRIBUTIONS

Evaluation of the design method and the tools with the case studies.
(Paper VII)
Chapter 2
Background

This chapter provides descriptions and necessary knowledge on the terms and concepts used in this thesis such as manycore architectures, software development tools, used programming models and languages, implemented streaming applications, and the environment used for the design method.

2.1 Manycore Architectures

The roots of parallelism in computers go back to 1960s. For instance, in 1962, Burroughs Corp. introduced the D825, a four-processor computer that accessed up to 16 memory modules via a crossbar switch [2]. Later, the first examples of modern parallel computers were developed with multiple components (processors, DSPs) each of which was on a separate chip. Even today we can see architectures consisting of combination of processors, GPUs, DSPs and FGAs, which are on separate chips. However, the form of parallelism that we focus on, multiple cores on the same chip, have started to appear around 2001 when IBM introduced POWER4 processor [86]. Soon after the appearance of the first multicore processor, single core processors reached thermal limits. Increasing the power consumption caused the chips become hotter and at some point cooling methods failed to maintain the chip temperature below critical degrees.

Mainly due to the thermal limitations, multicore processors became a trend after 2003. The first multicore processors were developed by just duplicating the fat processing core. Even today, this trend of having homogeneous processors with identical cores continues within the mainstream processors in personal computers. However, as the number of cores within the processor started to increase, new design options emerged. Cores, memories, and on-chip communication mechanisms are the main targets of different design configurations. The cores can be identical or vary widely. They can be coupled tightly or loosely e.g. they may or may not share the caches/scratchpads. They can be placed in clusters and each cluster might have its own shared memory. Core-to-core
Communication methods may vary by using message passing, shared memory models or both. Network-on-chip (NoC) might show variations in topology such as bus, ring, mesh, torus, crossbar, hypercube etc. Manycore chips may have shared, distributed or hybrid memory models. These models might have variations such as centralized shared memory, distributed shared memory, partly centralized partly distributed shared memory, etc. Some of these configurations are illustrated in Figures 2.1 and 2.2. Figure 2.1 [70] presents a manycore architecture, which consists of 1024 identical cores with distributed shared memory and 2 dimensional mesh NoC whereas Figure 2.2 [66] presents an architecture with two RISC cores, 12 vector processors, hardware accelerators, shared local memory and L2 cache, programmable interconnection and a single bus for on-chip communication. Our tool chain for the design method supports some of the aforementioned design configurations including size and number of cores, size and type of memories, and NoC type.

![Figure 2.1: Overview of Epiphany-V manycore architecture from Adapteva [70].](image)

The structure of the manycore plays a significant role in the performance while executing parallel applications, however, it is not the only actor. Two other major actors are the algorithm and the implementation. An application can be implemented as several parallel tasks running simultaneously to reduce the execution time. However, the execution time of the application cannot be shorter than the execution time of its longest sequential task. This effect is described by Amdahl’s law [45]. In the best case of parallelization, the speed-up factor can be as large as the number of cores. (The performance can be increased further by moving the entire application data from slower system memories to the caches if the application is divided into small enough tasks.)
However, achieving a speed-up factor that is close (or equal) to the number of cores is extremely difficult if not impossible. To be able to do that, the developer needs to put excessive amount of time and effort in re-factoring and optimizing the algorithm/implementation. On the other hand, tailoring an application for a specific architecture makes it difficult to be executed on other architectures due to the differences between the architectures. The difficulty of programming increases from single core to manycores and continues to increase when the manycores go from homogeneous to heterogeneous. More detail on this issue is given in the next section. In summary, manycore architectures improve the performance of processors [6] and decrease the power consumption [3], however, software productivity and portability decreases due to difficulties in parallel programming and differences in the characteristics of different architectures.

Despite the difficulties in designing and programming manycore architectures, they have a wide usage area from radars [89] to video coders/decoders [50], vehicles [75], hand-held devices, network devices, robotics [11] and many other cyber-physical systems. Many commercial and research architectures have been developed. Some notable commercial multi / manycore processors are Xeon Phi [21], Cell [51], big.LITTLE [37], Ambric [16], Epiphany [71], Myriad2 [10], TILE-Gx [61], SW26010 [93], and Xtensa LX [34].

In paper I and paper II, we have executed the case studies on the Epiphany architecture from Adapteva due to its high performance, low power consumption and floating point support. The architecture consists of 2D array of identical RISC cores. Each core has 32KB of local memory and access to a 32MB of external shared memory. The cores are connected to a mesh network, which
uses an XY algorithm for routing. Adapteva has released two different boards with 16 and 64 Epiphany cores. Even if Epiphany is a homogeneous architecture, we can use it to explore the mapping of streaming applications onto manycore architectures and as a baseline when we introduce heterogeneity.

2.2 Software Development for Manycores

Manycore architectures usually come with their own low level programming languages, and/or language extensions and libraries. In order to develop efficient parallel applications targeting these architectures, the developer needs good knowledge on parallel programming, the architecture and the supported language and probably the libraries. The difficulty of developing an efficient application increases with the complexity of the architecture. For instance, in homogeneous architectures, where the cores are identical, an application can be executed on any of the cores. Therefore, the developer can use a single language and a single compiler framework to create executables targeting these architectures. However, if the cores execute different sets of instructions, then the same binary may not be executable for each core. Thus, the developer might need to generate different binaries to be executed on different cores and this might cause using different compilation tools or even different languages. The developer might also be forced to know certain details about the architecture to figure out how to map the application on different hardware resources unless there is an efficient tool to do that. Since a single binary might not be executable for different cores, multiple, different binaries for the same task might be required to enable the task migration between cores. Having multiple binaries for the same task causes extra memory usage. There are other approaches to migrate tasks in heterogeneous architectures such as [49, 68, 80], however, we will not go through them as task migration is not the focus of this thesis.

In case the cores support different operations, the tasks might need to be modified to be executed on different cores. For instance a task, which uses floating-point operations, might be executed with no problem on a core that has a floating-point unit however, it might need to use fixed-point arithmetics when migrated to a core that does not have a floating-point unit.

High level programming languages with support for parallelism require less knowledge and less effort for parallel application development. In order to reduce the complexity of the development and execution of parallel applications, there is a need for languages with support for parallelism. However, there is no common language that is supported by the emerging manycore architectures. Hence, it is not possible to generate executables from a common language for these architectures by using the existing compilation tools. Therefore, there is a need for software development tools, which facilitate parallelizing the application, generating code for different target architectures, mapping tasks onto
the proper cores, etc. This is why software development is one of the main challenges for the manycore architectures on the path to become mainstream.

We do not address all of the software development and execution challenges in this thesis, however, we address some of them by evaluating the efficiency of a language with support for parallelism and a software development tool together with their impact on developer productivity and code portability.

In addition to programming languages and software development tools, programming model has a significant role in developing efficient software for manycore architectures. We have chosen dataflow programming model in order to implement our streaming applications, due to its inherent support for parallelism and streaming applications. The next section provides the details on this programming model.

2.3 Dataflow Programming Model

Dataflow programming [67] is a programming paradigm that models an application as a directed graph of the data flowing between operations. Dataflow model describes the computation in terms of distributed and locally controlled events, which correspond to ‘firing’ of actors. An actor is a computational entity, which consists of a single or multiple instructions and fires (starts the execution) when all the required inputs are available.

Streaming applications consist of a number of tasks, which are performed on a continuous flow of data. Dataflow programming model supports implementation of individual actors, which are computational entities consisting of single or multiple instructions. The tasks can be implemented as individual actors and these actors can pass the data to other actors to perform the next tasks. Multiple actors can execute simultaneously. These actors are connected through explicitly defined input and output ports. They can be locally synchronized by using blocking ports, which gives the streaming application the option of being globally asynchronous but locally synchronous. The ports are connected via channels. Figure 2.3 presents an application implemented with dataflow programming model.

A traditional application, that is implemented with dataflow programming model, is built as a series of actors connected in a specific order. For instance, in Figure 2.3 each box represents an actor whereas each arrow represents a unidirectional channel. There are several languages adopting this programming model such as Lucid [91], LUSTRE [38], StreamIt [87] and CAL actor language [26].

CAL is a modern dataflow language that is adopted by the MPEG Reconfigurable Video Coding (RVC) working group as part of their standardization efforts. We have used this programming language to implement our streaming applications, which will be described in coming sections.
CHAPTER 2. BACKGROUND

Figure 2.3: A sample streaming application - the IDCT2D block of MPEG-4 Simple Profile decoder; represented as interconnected actors (the boxes are the actors and the triangles are input/output ports, which can be connected to any source or sink). [79]

2.4 CAL Actor Language

The first step of our design method is application development. In order to facilitate parallel programming, we use a high level language with support for parallelism, that is the CAL actor language [26]. This language is an actor oriented dataflow language that extends Dataflow Process Network (DPN) [58] actors with states. A CAL program consists of actors and connections between these actors. A CAL actor consists of code blocks called actions, that can transform input data streams into output data streams, and input/output port(s) to connect to the other actors. CAL supports usage of parameters while instantiating these actors. This allows to create actors with different properties, private variables (to control the state of the actor), and named input/output ports (for communication with other actors and actions to perform a particular task). The state of an actor might change while transforming the input into the output. A CAL actor cannot access the state of other actors. The only way of interaction between actors is through input and output ports.

CAL actors perform computations by firing actions that has all the required conditions satisfied. In addition to the firing conditions of the DPN, which are the input value and the number of inputs tokens, CAL supports usage of actor’s internal state as a firing condition as well. Actors might have a scheduler that is a finite state machine (FSM) to order the firing of the actions. In the scheduler there might be more than one eligible action for firing. In such cases action priorities are used for selecting the action to fire. If there are no priorities
assigned to the actions and all firing conditions are satisfied for more than one action, the execution is non-deterministic.

**CAL** supports application development with different models of computation such as Kahn Process Networks (KPN) [32], Synchronous Dataflow (SDF) [57], Dataflow Process Networks (DPN) [58], and Cyclo-Static Dataflow (CSDF) [12].

As seen in Figures 2.4 and 2.5, an action may consist of several parts such as action name, input/output ports, action guard and action body. Name is a label for the action that can be used in the scheduler and while setting the action priorities. Input and output ports are the communication channels between actors. The actors exchange tokens through these ports. An action can read or write an arbitrary number of tokens by using the `repeat` keyword. Action guard is a conditional expression that must be satisfied to be able to run the body of the action where the input tokens can be consumed/processed, the output tokens can be produced and the state of the action can be changed. The input tokens are read at the beginning of the action, before the guard statements, however, consumed only at the beginning of the action body. The output tokens are produced at the end of the action body.

```
actor Split() int I1 ==> p
N:
 A1 : action I1[a] ==> P[a]
  guard a >= 0
  do a := a + 1; end
A2 : action I1[a] ==> N[a]
  guard a < 0 end
end
```

**Figure 2.4:** A simple splitting actor in CAL actor language.

```
actor Combine() int P, N ==> C:
 int z := 0
 action P:[a], N:[b] ==> C:[z]
  do z := a + b;
 end
end
```

**Figure 2.5:** A simple combining actor in CAL actor language.

The Split actor as seen in Figure 2.4 has a single input port, two output ports and two actions. The first action, **A1**, has two conditions to be fired. The first condition is the availability of the input on input port *I1* and the second
condition is the input being greater than or equal to zero. If these conditions are not satisfied, the next action will be checked for availability.

The actors can be connected using a Network Language (NL), which is used to create networks of actors. The actors and sub-networks are instantiated within the entities section and connected to each other within the structure section as seen in Figure 2.6. The network language supports generation of hierarchical networks and variable declarations. The variables are used to pass parameters to actor or sub-network instantiations.

\begin{verbatim}
network SimpleNL () In ==> Out:
entities
  split = Split();
  combine = Combine();
structure
  In     --> split.I1;
  split.P --> combine.P;
  split.N --> combine.N;
  combine.C --> Out;
end
\end{verbatim}

Figure 2.6: A network of two actors in CAL actor language.

2.5 Streaming Applications / Dataflow Programs

Streaming applications (or dataflow programs), as presented in Figure 2.3, consist of a chain of actors each of which apply certain operations to the data that flows through this chain. The actors communicate with each other through channels, which are usually uni-directional. These applications might use more than a single channel for communication between two particular actors and an actor can be connected to several other actors. The actors can be synchronized by blocking through communication. For instance, if there is no data on a channel, the actor on the receiving end might choose to wait for the actor that sends the data.

Wireless communication (massive MIMO), video coding/decoding, radar communication and other kinds of signal processing applications are suitable candidates to be implemented as streaming applications due to the similarities in the characteristics such as consisting of a chain of operations and performing these operations on a flow of input data. This thesis covers several streaming applications such as two-dimensional IDCT, different approaches for QR decomposition and autofocus criterion computation in SAR systems.

Within an application, the actors might have individual characteristics and require different hardware resources due to performing different types of com-
putations. For instance, the task that is performed within the *downsample* actor in Figure 2.3 is to forward every other input token to the output port, which does not require any computation power and consequently no special hardware resources. However, the *idct1d* actor consists of several arithmetic and logic operations and hence require arithmetic and logic units. Similarly, other actors might require other hardware resources such as floating point units, vector operators, high performance addressing units, larger memory banks, etc. Due to this inherent heterogeneity, streaming applications and heterogeneous architectures are a natural match.

Consisting of a chain of actors might be an advantage for these applications when it comes to mapping them onto manycore architectures. Unless the architecture has cores with different hardware resources and very efficient core-to-core communication mechanism, the actors can be mapped with a very simple heuristic algorithm such as mapping an actor onto an individual core and placing connected actors on neighbour cores. Otherwise, the communication pattern of the application, hardware requirement of the actors, and properties of the cores in the manycore architecture would be required for a proper mapping. So far, our tools generate one core per actor, and leaves the placement of the cores to the developer. The placement is done while instantiating the cores (with core IDs) and configuring the network-on-chip in the rocket chip generator. Automation of these tasks is a part of the future works.

In the next three sub-sections, we briefly describe the streaming applications implemented during the course of this thesis work.

### 2.5.1 Two Dimensional Inverse Discrete Cosine Transform (2D-IDCT)

Video is used widely in the human life for many purposes such as entertainment, communication, gathering intelligence, security, safety and control mechanisms. There are certain cases requiring high quality images or video stream. Therefore, large efforts were spent to create better quality images by increasing the resolution, size and colour depth. High quality images and videos required large amount of data space. Capabilities and complexities of encoders have been increased in order to store these images efficiently. Naturally, complexity of decoders have been increased as well. Consequently, performance of single core processors became insufficient for executing encoders and decoders.

Two dimensional IDCT, that is implemented as 15 actors, mapped onto 15 cores and presented in Paper I, is a part of MPEG-4 simple profile decoder [79]. The main usage area of this decoder is embedded systems with low latency. The decoder requires high computation power due to the low latency requirements, and capabilities and complexities of encoders in order to compress videos efficiently. The implementation consists of many actors with different characteristics e.g. some actors only shuffle data in a matrix, some controls memory accesses and some performs different types of computations. When
identical cores are used as in the Epiphany architecture, the cores, which perform duplicating or shuffling the data usually waits for the other cores, which perform heavier duties. Therefore some cores get underutilized.

2.5.2 QR Decomposition

QR decomposition (QRD) [33], also known as QR factorization, is used in several applications as a replacement of matrix inversions to avoid precision loss and to reduce the number of operations. The computation in the QRD is decomposition of a matrix into an upper triangular matrix R and an orthogonal matrix Q. We have implemented three basic approaches namely Givens Rotations, Householder transformation and Gram-Schmidt and presented in Paper II. The implemented actors are mostly identical, therefore executing them specifically on a heterogeneous architecture would not provide any specific benefits. This computation is a part of massive MIMO (multiple-input and multiple-output) applications [74].

Massive MIMO is a multi-user MIMO technology, which deploys very large number (hundreds or thousands) of service antennas in each base station and utilizes them in order to communicate with a number of (tens or hundreds) single-antenna terminals over the same time and frequency band.

2.5.3 Synthetic Aperture Radar Systems and Autofocus Criterion Calculation

Synthetic aperture radar systems [84] are usually mounted on moving platforms and uses the motion of the platform over a target region to create two or 3-dimensional high resolution images of landscapes. The path of the movement is not perfectly linear, however, this can be compensated with additional processing. The compensations are typically based on positioning information from GPS[88]. In cases where this data is insufficient, autofocus criterion can be used. There are different methods for calculating this criterion [18, 40]. One method tries to find the flight path compensation that results in the best possible match between two images of the contributing subapertures. This requires several flight path compensations to be tested. The matching is checked regarding a selected focus criterion. The criterion calculations, including interpolations and correlations, are performed many times. In paper IV, we present how we have developed hardware building blocks and an accelerator to perform these calculations efficiently.

As a conclusion, the streaming applications, which we developed, require high performance and low power. They consist of a chain of tasks with different characteristics and requirements. These tasks can be executed with different efficiencies on different types of cores. Therefore they are suitable candidates to be executed on heterogeneous architectures. However, each application would probably require different form of heterogeneity to be executed efficiently.
2.6 Evaluation of Software/Hardware Development Tools and Manycore Architectures

When designing manycore architectures and developing software and hardware development tools targeting these architectures, there is a need for a well structured evaluation method and assessment framework for all notable aspects—both hardware and software—of the development process. Such a method and framework are needed due to the new developments in hardware, such as the emergence of highly parallel architectures for energy efficient computing, and also in software for efficient code generation and code compilation.

We have identified four key areas of Productivity (in programming), Portability (of programs), Performance, and Power efficiency, where the introduction and careful use of appropriate measures can reduce the cost of development, and improve the quality of the final product.

In this thesis we focus on performance, productivity, and portability areas to evaluate the architectures and the software and hardware development tools. Performance evaluations are performed by measuring the execution times of the applications on different platforms in terms of clock cycles and seconds. Performance can be affected by both the algorithm/application and the architecture. Productivity evaluations are performed by measuring the time for developing software and hardware with and without software tools. It is mainly affected by the experience of the developer, software tools, programming models and languages. Software portability is highly related to re-usability of software components. By being able to reuse verified and tested code, the amount of maintenance (and engineering effort) can be reduced substantially. Portability can be evaluated by measuring the engineering effort for porting an application to a new platform.

2.7 Design of Manycore Architectures

There are three main structures in manycore architectures namely the processing elements, memory units and network-on-chip. The architecture gets different characteristics based on the configurations of these structures, mentioned in Section 2.1. Different combinations of these configurations form the design space of manycore architectures. Our design method aims to allow modification of these configurations to facilitate design space exploration of manycores with a focus on domain-specific heterogeneity through application specific accelerators.

Manual design of manycore architectures is an extremely challenging process. Therefore, we have used software&hardware development tools such as rocket chip SoC generator [5] and Cal2Many [31], and TURNUS [20] to automate our design method. Among these tools, we have extended the rocket chip generator and Cal2Many framework to support the design configurations that
we wanted to evaluate. Additionally, we have used Chisel language \cite{chisel} to de-
scribe the hardware, rocket custom co-processor (RoCC) interface to integrate
the accelerators to the processing core, which is the rocket core \cite{rocketchip}
based on RISC-V ISA \cite{riscv}. We chose RISC-V ISA mainly due to the support for cus-
tom instruction extension, which allows integration of accelerators to the base
cores. Being open-source, the developing team, supporting committee, and the
tools are the other motivations.

In the rest of this section we will provide the background information for the
aforementioned components, which are parts of the proposed design method.
The details of the design method and how these tools are used will be given
in Chapter 4.

2.7.1 RISC-V and Rocket Chip

RISC-V \cite{riscv} is a new and open instruction set architecture designed at UC
Berkeley to support research and education on computer architectures. The
base ISA consists of approximately 40 integer instructions and there is space
left to support extensions. Some of the standard extensions are multiply & di-
vide, atomics, single precision and double precision floating point instructions.
There is still opcode space left for non-standard extension to let the develop-
ers add new features to their processors. The ISA supports 32-bit, 64-bit and
128-bit addressing modes.

Rocket chip generator \cite{rocketchip} is an SoC generator written in Chisel, which
is an open source hardware construction language based on Scala \cite{scala}. The
generator consists of many parameterized libraries and can construct dif-
f erent manycore models based on RISC-V ISA. In the generator there are
six sub-components namely (I)core generator, (II)cache generator, (III)rocket
custom co-processor generator, (IV)tile generator, (V)tilelink generator and
(VI)peripherals. The generated tiles can consist of a core, memory and an ac-
celerator. There are two base types of cores supported by default, which are the
rocket core and the Berkeley Out of Order (BOOM) core \cite{boom}. The cache gen-
erator can generate both caches and scratchpads. It supports different scratch-
pad and cache sizes as well as various associativities and replacement policies
whereas the tile generator supports generation of multiple tiles with different
number of accelerators. The default on-chip communication structure is cross-
bar network. We extend this tool with a 2 dimensional mesh network-on-chip
to improve the scalability in terms of number of connected tiles.

2.7.2 Rocket core and RoCC Interface

The simpler (and smaller) of the two cores supported by the rocket chip gen-
erator is known as the rocket core \cite{rocketchip}. This core is an in order, single-issue,
scalar processor with a 5-stage pipeline. It executes the 64-bit RISC-V ISA and
features an integer ALU and an optional FPU. The default configuration of
this core includes first-level instruction and data caches. Additionally, the core features an interface called rocket custom co-processor (RoCC) to integrate co-processors/accelerators.

It is this RoCC interface we will use to incorporate our application specific accelerators for our heterogeneous manycore systems. This interface provides connections to the core and memory. Custom instructions are then used for interaction with the accelerators through this interface. These instructions provide two source registers and a destination register together with some extra bit fields. A bit field in the instruction indicates if the rocket core requires an answer from the accelerator. Figure 2.7 gives an overview of the request and response connections between the core and the accelerator. The source registers signals ($rs$ in the figure) in the request interface can be used to transfer data from core to the accelerator and whereas the destination register signal ($data$ in the figure) in the response interface can be used to return results from the accelerator to the core. The $inst$ signal carries the fields of the custom instruction that is bound to the accelerator. The details of the custom instruction are given in Chapter 4.

![Diagram of RoCC interface](image)

**Figure 2.7:** Connections between the core and the accelerator provided by the RoCC interface. Request and response connections are separated for a clearer illustration.

The interface provides an interrupt line and connections between the accelerator and the memory as well. The memory connections are not required in our implementations in Paper IV, and thus they are not used. However, in Paper VII, the accelerators in one of the case studies use the memory connections to read data directly from the memory.

### 2.7.3 Chisel

The Chisel [8] language is an open-source hardware description language developed at UC Berkeley to support advanced hardware design and circuit generation. The rocket chip generator is developed in this language. Chisel uses
FIRRTL as an intermediate hardware representation language and it is embedded in the Scala language. In reality Chisel is only a set of class definitions, predefined objects, and usage conventions within Scala. A Chisel program is actually a Scala program that constructs a hardware graph. The main features of the Chisel language are parameterized generators, algebraic construction, abstract data types, bulk connections, hierarchical, object oriented and functional constructions and multiple clock domains. The Chisel compiler, which is provided by UC Berkeley, can generate low level synthesizable Verilog code.

There are several datatypes supported by Chisel. A raw collection of bits is represented by the Bits type. Signed and unsigned integers are considered subsets of fixed-point numbers and are represented by types SInt and UInt respectively. Boolean values have the type Bool. These types are distinct from Scala’s built-in types such as Int or Boolean. Constant or literal values are expressed using Scala types and Chisel types are used for expressing the values of the wires and registers.

Additionally, Chisel defines Bundles for making collections of values with named fields (similar to structs in other languages), Vecs for indexable collections of values and Mem to generate random access memories.

Figure 2.8 presents a 2to1 multiplexer hardware written in Chisel.

```scala
class Mux2 extends Module {
  val io = new Bundle{
    val sel = Bool(INPUT)
    val in0 = Bool(INPUT)
    val in1 = Bool(INPUT)
    val out = Bool(OUTPUT)
  }
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```

**Figure 2.8:** 2to1 multiplexer hardware written in Chisel.

Chisel supports modules, which are very similar to Verilog modules. Users define modules as classes that inherits from Module as can be seen in Figures 2.8. A class contains an interface wrapped in a Module’s IO() method and stored in a port field named io, and wires together sub-circuits in its constructor. Chisel has many other features, which can be found in [1, 8].

We developed the 2 dimensional mesh network-on-chip router in Chisel to integrate it to the rocket chip generator. Additionally, we generate custom hardware accelerators in Chisel as well, as they are to be integrated to the rocket chip generator. The accelerator generation is automated with the Cal2Many framework [31] that is described in the next section.
2.7.4 Cal2Many

The second step of our design method is analysis and code generation. In order to generate application specific hardware accelerators and target specific software code, we use the Cal2Many framework [31] that is developed in-house. This framework is a source-to-source compilation framework that takes a CAL program and generates an implementation expressed in the native language of the target architecture. In its original form, the framework supported only software code generation. However, with the back-end that is developed during the course of this thesis work and presented in paper IV, hardware code generation is supported as well. This back-end is extended in the work that is presented in paper VII, in order to support generation of multiple accelerators and parallel software code with a communication mechanism.

The code generation is performed in three steps. First, each CAL actor is translated to an actor machine (AM) [48] intermediate representation. Then this representation is translated to another intermediate representation called Action Execution Intermediate Representation (AEIR) [31]. Finally, the AEIR and the network description of the application are used by different back-ends to generate target specific source code. This operations are illustrated in figure 2.9. The highlighted back-end is the one that generates both hardware and software and added within this thesis. The library in the figure represents the communication library that is used for the Epiphany architecture, the hardware library for arithmetic operations, and possible future libraries. The map box represents the mapping information on how the CAL actors should be mapped onto the target architecture.

Actor machine is a controller with a set of states generated by combining the firing conditions of actions with the finite state machines (scheduler) and the action priorities. The AM states have knowledge on the conditions and a set of AM instructions that can be performed on the states. These instructions are test to test a firing condition, exec to execute an action and wait to change the state of an input token from absent to unknown so that a test on the input port can be performed later.

In order to execute an AM, its constructs must be transformed to different programming language constructs. For instance, AM instructions can be transformed into function calls whereas the transitions between the AM states can be controlled with if statements. However, these constructs might have different implementations in different programming languages and platforms. In order to abstract these constructs and bring the AM closer to an imperative sequential action scheduler without having to select a target language, the AEIR is introduced. The translation of the AM to the AEIR consists of two main task. The first task is the translation of the CAL constructs to imperative constructs including actions, variable declarations, functions, statements, and expressions. The second task is the generation of an action scheduler (a
state machine) by translating the states and the instructions of the AM to imperative language constructs.

The hardware generation is performed for the compute intensive actions of the \texttt{CAL} application. A \texttt{C} function that executes the custom instruction is generated for this action along with the hardware accelerator that performs the computation. The hardware generation is performed in two phases. In the first phase an SSA form is generated from the AEIR structure and some other information is collected such as the inputs and outputs of the accelerator. In the second phase the SSA form is translated into \texttt{C} and \texttt{Chisel} code. The details of this back-end is given in Chapter 4 and paper IV.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cal2many_overview}
\caption{An overview of the Cal2Many framework.}
\end{figure}
2.7. DESIGN OF MANYCORE ARCHITECTURES

2.7.5 TURNUS

Cal2Many framework generates hardware accelerators for the most compute intensive CAL actions. However, these actions must be identified first. In order to identify the compute intensive actions, we profile the CAL application using the TURNUS framework [20]. TURNUS is a framework for profiling streaming applications running on heterogeneous parallel systems. The framework uses the Open RVC-CAL compiler (ORCC) [95] as its base and extends its functionality with a dataflow profiler. It is integrated in the Eclipse environment and it provides abstraction level for application and architecture modeling. Additionally, it includes tools to profile and optimize the application, and estimate the system level performance.

The tool provides static and dynamic analysis information on both the software and the hardware features. A set of heuristics (e.g. critical path analysis, communication buffer minimization and optimization, partitioning and scheduling analysis) are available for optimization and exploration of design space of dataflow applications. However, we focus on the software aspects of the analysis and use the bottleneck analysis data to identify the compute intensive sections of programs. The bottleneck analysis depends on the firing rates of the actions, number of the executed operations, number of the operands for each operation, and manually entered weights for each operation. These weights are determined by the developer based on the target architecture. In our implementations, we use the required clock cycles for each arithmetic operation as their weights. The bottleneck analysis provides a final weight for each action in each actor showing an estimation of the execution time (in abstract units based on the weights of the operations). In order to obtain this dynamic analysis data, the applications must be simulated at least once.

The tool can provide performance estimation based on an architecture model that can have different computation, communication and memory components. So far we do not want to limit the hot-spot identification to any particular architecture model. However, in the future, for more accurate analysis and identification, one can use a complete model of the base core including the weights for all operations—not only the arithmetic ones. This would give a better understanding on how the application would behave on the target core and whether it would be beneficial to use any accelerator.
Heterogeneity in the context of computer systems usually means combination of processors with different instruction sets. However, there are different levels of heterogeneity and each level has advantages and disadvantages. One typical usage of heterogeneity is when an ordinary CPU is combined with a digital signal processor (DSP), specialized on fast multiply and add operations found in most signal processing applications. Thus, heterogeneity has for a long time been used as a way to accelerate computations in multiprocessors. Now, in the era of manycore architectures, we visit heterogeneity again in order to accelerate computations and reduce power consumption.

In parallel applications, especially in dataflow (a.k.a. streaming) domain, each core might perform a different task, and each task might require different computation, communication and memory resources for efficient execution. Our case studies on Epiphany architecture, which is a homogeneous architecture, have revealed that some cores might become a bottleneck due to executing tasks with specialized or higher requirements than the other tasks. Additionally, a sequential task, that is supposed to be executed on one core, might be too large to fit into the core. Dividing the sequential task on several cores and dealing with data dependencies would more likely harm the performance. All these facts imply that homogeneity is not the best solution and we need different types of cores to execute different parts of applications which means there is a need for introducing heterogeneity in manycore architectures. Moreover, our results in paper IV and paper VI and a number of other research studies [56, 15, 60] have shown that heterogeneous architectures are more efficient than homogeneous architectures in power consumption and performance when implemented on the same die size.

Besides the power and performance advantages, heterogeneous architectures have drawbacks. These drawbacks can be divided into two aspects namely productivity (engineering efficiency) and portability (task/application migra-
tion both on-chip and off-chip). In order to make use of all resources in a heterogeneous architecture, one needs to develop different code for different cores unless there are software tools which generate specific code for each core. This requires knowledge and effort and decreases the productivity of the developer.

In case of under-utilization of the resources such as cores, the tasks can be migrated from one core to another and the unutilized core can be shut down to save power. However, migrating tasks in heterogeneous architectures is a challenging duty. The cores might execute different ISAs which would require storing different binaries for the same task to support the migration. In some cases the new core might not support the operations performed by the task which might result in not migrating the task.

Another problem is the lack of knowledge about heterogeneity requirements of applications. This problem requires defining different forms of heterogeneity, profiling the applications, and exploring architectural design space of heterogeneous manycores. In order to address the first requirement, we define different levels of heterogeneity in the next section.

### 3.1 Levels of Heterogeneity

In recent years we have seen the introduction of new forms of heterogeneity as a means to improving energy efficiency. One such example is the big.LITTLE concept from ARM, where two processors with the same ISA have different implementations in order to allow for a trade-off between performance and power-consumption. In a way the big.LITTLE concept can be seen as a "mild" form of heterogeneity. There are many other forms (or as we will call them, 'levels') of heterogeneity. They range from lightly heterogeneous architectures with processors using the same ISA but with different implementations [54], each resulting in different performance and power usage, all the way to very heterogeneous architectures with a large number of customized processing elements with different ISAs utilizing dark silicon concept [28] to reduce power consumption and manage thermal issues [36]. With these different forms of heterogeneity we find it natural to introduce the concept of Levels of Heterogeneity.

We base heterogeneity levels on the number of complete ISAs executed by the architectures. The level is equal to the number of ISAs with one exception. Homogeneous architectures execute one ISA, however, we consider them as Level 0 and leave Level 1 for the heterogeneous architectures with one ISA. If a core in a Level 1 architecture supports more instructions than the other cores, the heterogeneity level becomes Level 1Aug to reflect that some cores execute augmented version of the base ISA. This applies to the other levels as well.

In our definition we focus on the heterogeneity of the cores, and even if one can foresee architectures with heterogeneous memory models, this aspect
3.1. LEVELS OF HETEROGENEITY

has not been included. Based on this description we can identify the following levels of heterogeneity:

- HetLevel0: Level 0 contains homogeneous architectures.
- HetLevel1: Level 1 contains architectures with a single ISA but multiple implementations (IMPs).
- HetLevel1Aug: Level 1Aug contains the architectures where all the cores share a single base ISA but have different instruction augmentations.
- HetLevel2: Level 2 contains the architectures which contain two different ISAs (and therefore at least two distinct IMPS).
- HetLevel2Aug: Level 2Aug contains architectures where the cores execute two base ISAs and have different instruction augmentations.
- HetLevelN: Level N contains the architectures which contain N different ISAs.
- HetLevelNAug: Level NAug contains the architectures where the cores execute N base ISAs and have different instruction augmentations.

At Level 0 we have all the classical homogeneous parallel computers where all cores are identical. These multicore computers are today found in everything from smart phones to high-end desktop computers. As on-chip versions of symmetric multiprocessing (SMP) systems, which have decades of research behind them, these multicore architectures are well understood and have a relatively straightforward programming model.

In Level 1 every core executes the same instruction set. However, the implementation is not the same for all cores. Each core consists of different functional units and control logic while achieving different performances and energy efficiencies on the same application due to having different performance and power characteristics. Additionally, the cores might have differences in the length of pipelines, execution type (superscalar, in order, out of order), structure of functional units (parallel units) etc.

The simplest implementation would be having two different cores such as the big.LITTLE [37] architecture of ARM. However, there is no limit on the number of different cores. In our definition, any architecture with cores which differ in implementation but run the same complete ISA is a Level 1 architecture. There are many similarities between the HetLevel1 architectures like big.LITTLE and HetLevel0 architectures using DVFS and they can potentially be treated in the same operating system framework.

In some cases, an architecture might have different types of cores executing the same instruction set however some of the cores might support additional instructions due to having additional functional units such as floating point units or digital signal processing extensions. When an application does not
need these extensions, it is executed on the cores without the extensions to consume less power. In such architectures, we consider the ISA of the core, which has additional units, as an augmented ISA, and the heterogeneity level of such architectures as Level 1aug.

Level2 heterogeneous manycores consists of cores which use two different instruction set architectures. The implementations of the cores are different by default due to executing different ISAs. There are many architectures like the Epiphany which has a host core and an array of identical accelerator cores residing on different chips and executing different ISAs. This kind of architectures look like Level 2 architectures however, we do not include them in our definition due to having different types of cores on different chips. These architectures can become Level 2 architectures by moving the host core and the accelerator cores on to the same chip.

HetLevelN covers the levels above two without any ISA augmentation whereas HetLevelNAug covers the levels above two with augmented ISAs. There currently are no commercial architectures with three or more complete ISAs. However, we believe there will be such architectures in the future. Especially mobile platforms would gain from the energy efficiency of these architectures.

Table 3.1 shows the main levels of the heterogeneity together with example commercial architectures. Second and third columns are presenting the number of instruction set architectures and number of different implementations respectively.

<table>
<thead>
<tr>
<th>Level</th>
<th>#ISA</th>
<th>#IMP</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>1</td>
<td>1</td>
<td>Epiphany,</td>
</tr>
<tr>
<td>Level 1</td>
<td>1</td>
<td>2+</td>
<td>big.LITTLE, TILE-Gx</td>
</tr>
<tr>
<td>Level 1Aug</td>
<td>1+Aug</td>
<td>2+</td>
<td>Xtensa LX, Ambric</td>
</tr>
<tr>
<td>Level 2</td>
<td>2</td>
<td>2+</td>
<td>Cell, Myriad2</td>
</tr>
<tr>
<td>Level N</td>
<td>N</td>
<td>N+</td>
<td></td>
</tr>
</tbody>
</table>

3.2 Conclusion

We defined heterogeneity in manycores and classified the heterogeneous architectures. We presented advantages and drawbacks of these architectures based on their levels. We conclude that heterogeneous architectures have the potential of achieving higher performance and lower power consumption when compared to homogeneous architectures. However, the advantages do not come for free. There are drawbacks in terms of productivity and portability which
3.2. CONCLUSION

we believe can be solved (up to some extent) by developing software development tools. Additionally, there is still a huge design space to be explored to make best use of heterogeneity.
Chapter 4
Domain-Specific Heterogeneous Manycore Design

Specialized single core processors and heterogeneous manycores have been studied and proven to be more efficient than general purpose processors [56, 15, 60]. Additionally, authors of [42] suggest that the future of the computer architectures is the domain-specific architectures. However, introducing specialization and heterogeneity in manycores increases the complexity of the architecture. Hence they are more difficult to develop and program when compared to homogeneous architectures.

In this chapter we describe a methodology to ease and facilitate development of heterogeneous manycore architectures. There have been several studies, which try to facilitate development of manycore architectures and explore the design space [17, 94, 53, 83] using simulation tools [13, 73, 78, 62, 19, 30]. However, the design space for manycore architectures is huge, especially when heterogeneity is involved. Exploring all heterogeneity levels to find the most suitable architecture for the target applications would require significant amount of time and effort. On the other side, simulations might be too long to be feasible when the size of the simulated systems increase to hundreds of cores [96]. More importantly, the simulation tools do not provide the hardware resource usage requirements and achievable clock rates.

There are frameworks such as McPat[59] and OpenPiton [9], which aim to generate manycores for FPGA and ASIC synthesis and provide hardware related results. However, these frameworks provide limited configurations that limits their applicability to cover the whole design space. OpenSoC System Architect [29] is another framework that can generate manycore architectures based on RISC-V instruction set architecture. It provides a richer set of configurations including custom accelerator extensions. The developer has to manually determine the configurations and the custom accelerators and then implement these accelerators using a hardware description language.
In order to shrink the size of the design space, we propose to design the architectures based on the requirements of the applications within a target domain. There have been processors designed for specific applications [82, 46, 35], however we target a more flexible architecture, which can cover a domain with many application specific cores. Moreover, the method that we propose is generic and can be used for development of different architectures targeting different application domains. We divide this method into 4 main steps namely, application development, analysis and code generation, accelerator integration, and system generation as seen in Figure 4.1. This figure gives a generic view of the design method.

Since the aim is to develop application and domain-specific architectures, the design method starts with the development of the application. The programming model and the language should expose parallelism and hide the details of target architectures in order to simplify the programming. Once the application is developed, it requires to be analyzed to identify the computationally intensive parts (hot-spots). An application may have multiple hot-spots or the hot-spots may not be clearly visible. This information can be used by the developer to modify the application. After identifying the hot-spots, custom hardware (accelerators) should be developed to perform the operations of these hot-spots. However, first, the base core must be chosen to determine the hardware description language and the interaction protocol between the core and the accelerators. The core should preferably be small as it will delegate the computations to the accelerators and need to support custom hardware extensions. It should be possible to have multiple hot-spots converted into multiple accelerators, however, the target core may have limitations on this aspect. If there are no visible hot-spots in the application, either the developer edits the application, or there will be no accelerators for the application. The rest of the application should be converted into a language supported by the native compiler of the target core. The converted code needs to take care of the interaction with the accelerator(s).

The applications may require a number of cores with different number of accelerators, varying amount of memory, and other hardware resources. These requirements can be used as configuration parameters while configuring the base core and integrating the accelerators. Each core may be configured individually, or several cores may have the same configuration based on the requirements of the application. After configuring the cores and integrating the accelerators, there is a need for a network-on-chip (NoC) to connect the cores together. A parallel application may have parts, which require sending or receiving data to/from other parts of the program. Therefore, a communication mechanism is necessary. The NoC should be scalable to enable using different number of cores for different applications. When the cores are connected with a NoC and the final architecture is developed, the software code needs to be compiled and mapped on to the cores.
Figure 4.1: A generic view of the design flow for developing domain-specific heterogeneous manycore architectures.
Using the requirements of the applications or application domains and specializing the architecture shrinks the design space, however, developing the architectures is still an immense challenge. In order to remove the burden of developing an architecture from scratch, we use software tools to automate the steps of the design method. Some of these tools are developed within this thesis and some of them are developed by others. Figure 4.2 shows a realization of the design method with the tools, languages and other components. We will go through the steps of the design method again, but this time with description of realization and the tools shown in the figure.

4.1 Application Development

We have chosen the dataflow programming model and CAL actor language to develop the applications due to their suitable characteristics, which are given in the background chapter, for parallel programming. The development of the CAL applications is done in Eclipse environment, where the ORCC framework can be integrated as a plug-in. This framework provides simulation for CAL applications. The verification of the CAL applications is done through simulation with the ORCC framework. During the development, the restrictions and guidelines of the code generation tools must be followed. For instance, the code generation tool, which will be described in the next section, has rules on defining constants and using certain operations such as convolution and square root.

Figure 4.3 shows an action of QR decomposition developed in CAL. The square root operation is placed in a procedure called SquareRoot and the result is read from a variable called SquareRoot_ret. These names are defined keywords for the code generation tool. The rules apply only if the action is a hot-spot and a custom hardware will be generated for it. Additionally, if an action is identified as a hot spot, __acc__ needs to be added to its name as a prefix to be recognized by the code generation tool.

When the application is developed and the functionality is verified, it is fed to the analysis and code generation tools.

4.2 Analysis and Code Generation

The application code is first fed to the analysis tool, i.e, the TURNUS framework. This tool analyzes the code and provides various information on software and hardware aspects. We only need the bottleneck analysis of this tool. In this analysis, the tool counts the firing of each action and the number of each operation type. Then it applies a set of weights defined by the developer to the operations to estimate the contribution of each action to the execution time of the actors. An example of the bottleneck analysis can be seen in Figure 4.4, where the Weight corresponds to the contribution of each action to the execution time. According to the figure, 49.48% of the execution time of the
Figure 4.2: Realization of the design flow for developing domain-specific heterogeneous manycore architectures.
CHAPTER 4. DOMAIN-SPECIFIC HETEROGENEOUS MANYCORE DESIGN

Figure 4.3: An action of QR decomposition application written in CAL actor language.

```
calculate_boundary : action =>
guard row_counter < COL_SIZE
var
    float a,
    float b,
    float r_tmp,
    int index
do
    index := row_counter * ROW_SIZE + col_counter;
    if x_in[row_counter] = 0.0 then
        c := 1.0;
        s := 0.0;
    else
        index := row_counter * ROW_SIZE + col_counter;
        a := r[index] * r[index];
        b := x_in[col_counter] * x_in[col_counter];
        SquareRoot(a + b);
        r_tmp := SquareRoot_ret;
        c := r[index] / r_tmp;
        s := x_in[col_counter] / r_tmp;
        r[index] := r_tmp;
    end
end

col_counter := col_counter + 1;
```

main actor is spent on executing the cubic action, out of 20 actions. Hence, this action is a proper candidate for acceleration. The TURNUS framework is integrated into the Eclipse environment like the ORCC framework.

The application developer might want to optimize the the action, that is identified as the hot-spot, to have only computations in the action and generate more efficient accelerators. Or the developer might need to edit the hot-spot action due to the restrictions in the hardware code generation. For instance, the hardware generator does not support mixture of integer and floating-point data types in the same action. Hence, the last assignment in Figure 4.3 (col_counter := col_counter + 1) is not supported. Either the assignment will be removed from the action, or the data type of the col_counter will be changed from int to float. Other restrictions are; loops are not supported in the hardware, array dimensions need to be variables declared with literal assignments, and constants should be assigned only literals.

When the hot-spots are identified and marked with the prefix (__acc__), the CAL code is fed to the code generation tool, i.e., the Cal2Many framework.
4.2. ANALYSIS AND CODE GENERATION

Figure 4.4: Bottleneck analysis results of Autofocus criterion calculation application implemented in CAL, provided by TURNUS.

The C/Chisel hybrid back-end is used to generate the hardware/software co-design. The framework converts the CAL application into actor machine (AM) model and then into actor execution intermediate representation (AEIR) as described in Section 2.7.4. This structure includes actor list, action lists, port lists, functions, guard conditions, scheduler, variable declarations, statements, etc. The back-end goes through the AEIR structure and generates C code for the actions except the marked (hot-spot) one. When the `__acc__` prefix is found in an action name then both Chisel and C code are generated for that action. First, the hardware code is generated by going through the AEIR structure for the action twice. In the first iteration static single assignment (SSA) form of the action is generated in order to avoid re-assignments in the hardware. Additionally, the global variables, which are read within the action are marked as inputs and the ones that are modified are marked as outputs. In
the second iteration the corresponding Chisel code is generated for each I/O port, variable, and statement. The arithmetic operations including addition, subtraction, multiplication, division, square root, and $7 \times 7$ convolution are replaced by the pre-defined hardware modules, which support IEEE single-precision floating-point formats and are stored in the hardware library. The division and square root hardware module implementations are based on a novel method [44, 43]. The details of these modules are given in paper III and paper V, respectively.

The accelerator is integrated to the core as a custom instruction. Hence to be able to interact with the accelerator, the custom instruction must be embedded into the C code. After generating the hardware code for the hot-spot action, the hybrid backend generates the same action in C. However, the body of the action gets replaced with custom instruction call(s). The number of instruction calls depend on the number of input and outputs. Each instruction has two source register fields as seen in Figure 4.5. The register size is 64 bits. When two floating-point numbers (each 32 bit) are combined in a source register, four floating-point numbers can be transferred to the accelerator with a single instruction call. The destination register is 64 bit as the source registers. Additional instruction calls can be used to read multiple results from the accelerator. The \texttt{xd} field of the instruction determines if the core will wait for a response from the accelerator. If the bit is set, the core will halt all operations and wait for the result, otherwise it will continue executing instructions. We use the \texttt{funct} field to identify the instruction call as \texttt{read}, \texttt{write} and \texttt{fire}. The \texttt{write} function is used for transferring the data to the accelerator. The \texttt{read} function is used for reading additional results from the accelerator and the \texttt{fire} function is used to tell the accelerator that all the input data is transferred and it is ready to start the computations. In case the input data is an array, the core sends the address of the array and the size of the data to the accelerator. The accelerator needs to be connected to the memory manually to read the data directly from the memory. This method is faster than transferring the data with the instruction calls when the data is in an array.

\begin{figure}[h]
\centering
\begin{tabular}{cccccccc}
\hline
31 & 25 & 24 & 20 & 19 & 15 & 14 & 13 & 12 & 11 & 7 & 6 & 0 \\
\hline
\texttt{funct} & \texttt{rs2} & \texttt{rs1} & \texttt{xd} & \texttt{xs1} & \texttt{xs2} & \texttt{rd} & \texttt{opcode} \\
7 & 5 & 5 & 1 & 1 & 1 & 5 & 7 \\
\texttt{src2} & \texttt{src1} & \texttt{dest} & \texttt{custom} \\
\hline
\end{tabular}
\caption{Bit fields of the custom instruction of RISC-V instruction set.}
\end{figure}

The code below (Listing 4.1) shows the macros we use to call the custom instructions. The variables ending with \texttt{pos} and \texttt{val} are 64-bits and copied
4.3. ACCELERATOR INTEGRATION

into the source registers. The first instruction call is a non-blocking one, where the xd bit field is set to zero and the funct field is set to write (FUNCT_IN1 in the code). The second instruction call is a blocking one with the fire function to trigger the accelerator to start the computations. The core halts until the accelerator returns a response. Finally, when the response is ready, the result is read from the destination register into a variable named outputReg.

```
// Lets execute the custom instructions
uint64_t outputReg;
ROCC_INSTRUCTION_NO_BLOCK(XCUSTOM_ACC, outputReg, 
  beamer_1_complex_input1_p_pos, 
  beamer_1_complex_input1_p_val, FUNCT_IN1);
ROCC_INSTRUCTION(XCUSTOM_ACC, outputReg, 
  beamer_1_complex_input2_p_pos, 
  beamer_1_complex_input2_p_val, FUNCT_FIRE);
// Lets read the results
beamer_1_complex_output_p = outputReg;
```

Listing 4.1: Custom instruction calls in C.

The C code generation includes a simple core-to-core communication mechanism based on flags and FIFO buffers. It performs only remote writes due to remote reads being expensive. The connections between the CAL actors are converted into channels. Each channel has a valid flag and a buffer in the receiver core and a ready flag in the sender core. The sender checks the ready flag to write data into the buffer that is on the receiver core’s local memory. This operation is blocking. The core will wait until the ready flag is set before writing any data. Once the sender writes the data to the receiver’s buffer, it sets the valid flag and resets the ready flag. The current buffer size is 1 floating-point number. However, this can be changed in the future to support dynamic buffer sizes based on the requirements of the application. The receiver checks the valid flag before reading any input data from the buffer. If the flag is not set, then the core will block until it is set. When the core reads the data from the buffer, it resets the valid flag and sets the ready flag. These operations do not include any reads from any remote memory. All the read accesses are only to the local memories.

4.3 Accelerator Integration

We have chosen the rocket core that is generated by the rocket chip generator as the base core. The core has an interface called rocket custom co-processor (RoCC) to support custom hardware as a custom instruction that is already supported by the RISC-V gcc compiler. The core, the chip generator and the interface are described in the background chapter.

Integrating the accelerator to the rocket chip generator is performed in two steps. First the hardware implementation is copied into the source folder of the
rocket chip generator. Then a new class that extends the RoCC class is created to instantiate the accelerator and establish the necessary connections between the accelerator and the core and between the accelerator and the memory.

The following code snippet (Listing 4.2) shows an example of the connections between the accelerator and the memory. The connections are established in the class that extends the RoCC interface. The top part of the code shows a read request that is being sent to the memory and the bottom part shows how to read a memory response.

```
// Send read request to the memory
when(readMem){
  io.mem.req.valid := true.B
  io.mem.req.bits.addr := patchAddress + ((colIndex + (rowIndex * 230.U)) <<< 2)
  io.mem.req.bits.tag := memReqCounter
  io.mem.req.bits.cmd := M_XRD // perform a load (M_XWR for stores)
  io.mem.req.bits.typ := MT_W // D = 8 bytes, W = 4, H = 2, B = 1
  io.mem.req.bits.data := Bits(0) // we're not performing any stores
  io.mem.invalidate_lr := Bool(false)
when(io.mem.req.ready){
  memReqCounter := memReqCounter + 1.U
  colIndex := colIndex + 1.U
}
.otherwise{
  io.mem.req.valid := false.B
}
}

// Read the response from the memory
when(io.mem.resp.valid){
  convAcc.io.conv1Inst_filterCounter_in := filterCounter + memRespCounter
  convAcc.io.conv1Inst_inputImage_in := io.mem.resp.bits.data
  convAcc.io.readInput := true.B
  memRespCounter := memRespCounter + 1.U
}
```

Listing 4.2: Memory connections in the extended RoCC interface.

When the interface extension is completed, the new interface can be instantiated, while configuring a core in the configuration files of the rocket chip generator. The instantiation is done by binding the interface to a custom instruction as follows in Listing 4.3.
rocc = Seq(RoCCParams(
  opcodes = OpcodeSet.custom0,
  generator = (p: Parameters) => {
    val cubicAcc = LazyModule(new GeneratedCubicComplexAccelerator()
     ()(p))
      cubicAcc})
)

Listing 4.3: The instantiation of an accelerator that is bound to a custom instruction.

The code instantiates the GeneratedCubicComplexAccelerator interface and names it as cubicAcc. It then binds this interface to the custom instruction called custom0. Whenever the core executes the custom0 instruction, it will be forwarded to the RoCC interface. The interface supports up to four of these instructions by default.

The next step is configuring the cores and generating the full system.

4.4 System Integration

The rocket chip generator supports generation of multiple cores in a system. However, the default communication mechanism is a crossbar network. Figure 4.6 illustrates a crossbar network for 8 cores. This network is expensive to scale due to resources used for each core and the length of the wires. Therefore, we integrated a scalable two dimensional mesh network-on-chip to the rocket chip generator. Figure 4.6 illustrates the two dimensional mesh network as well. The details of this integration are given in Paper VII.

![Diagram](attachment:image.png)

Figure 4.6: A crossbar network with 8 cores and 16 switches on the left and a two dimensional mesh network with 16 cores and 16 routers on the right.

There are few configurations needed in order to generate a full system with the rocket chip generator. The first configuration is performed during the instantiation of the cores. Each core needs to be configured with many parameters including FPU usage, virtual memory usage, icache size, scratchpad size,
core id, accelerator usage, and bus sizes. A core configuration can be copied to instantiate several cores with the same configuration. After configuring the cores, the network-on-chip needs to be configured with the row and column size of the mesh and the size of the local scratchpad memories. By default, the current implementation assumes a single size for all of the scratchpad memories. However, this can be changed to support different memory sizes for different cores by configuring the network interface. A NoC router is generated for each core unless the architecture is a single core. The connections between the routers are established automatically.

The code in Listing 4.4 is an example of a single core instantiation with the aforementioned configuration parameters.

```scala
val core0 = RocketTileParams(
  core = RocketCoreParams(
    useVM = false,
    fpu = None,
    mulDiv = Some(MulDivParams(mulUnroll = 8)),
    btb = None,
    icache = Some(ICacheParams(
      rowBits = site(SystemBusKey).beatBits, 
      nSets = 64, 
      nWays = 1, 
      nTLBEntries = 4, 
      blockBytes = site(CacheBlockBytes)),
    
    dcache = Some(DCacheParams(
      rowBits = site(SystemBusKey).beatBits, 
      nSets = 4096, // 16*16kb 
      nWays = 1, 
      nTLBEntries = 4, 
      blockBytes = site(CacheBlockBytes), 
      scratch = Some(0x80000000L)), // use scratchpad
    
    hartId = 0,
    rocc = Seq(RoCCParams(
      opcodes = OpcodeSet.custom0,
      generator = (p: Parameters) => {
        val cubicAcc = LazyModule(
          new GeneratedCubicComplexAccelerator().(p))
        cubicAcc
      })
  )
)
```

**Listing 4.4:** The instantiation of a rocket core within rocket chip generator.

The instantiated core does not have virtual memory, floating-point unit, and branch target buffer. The instruction cache size is 4096 bytes, the data memory is a scratchpad with the size of 256 KB and its address range starts from 0x8000000. The core id is 0 and there is an accelerator bound to custom0 instruction.

When all the cores are instantiated within the configuration class, the final architecture can be generated by using a Scala program that invokes the
Chisel compiler to produce RTL for describing the complete system-on-chip (SoC). Additionally, another Scala program can be called to generate cycle accurate C++ emulator. We use the emulator for performance measurements and the RTL implementation for hardware resource usage and clock rate measurements.
Chapter 5
Summary of the Papers

5.1 Evaluation of Software Development Tools for Manycores - Paper I

Developing applications for manycore architectures is a challenging task. The developer must have advance knowledge about the architecture and the language that is supported by the architecture. These languages are usually low level and include architecture specific extensions. In order to decrease development complexity and knowledge requirement, high level languages and code generation tools are developed. CAL actor language is a high level language aimed for the usage of parallel application development and Cal2Many [31] is a compilation framework which takes CAL code as input and generates compilable code for several different manycore platforms.

The contribution of this paper is the evaluation of the CAL actor language together with the Cal2Many compilation framework. Based on the evaluation and feedback, the compilation framework has been optimized. We developed two dimensional inverse discrete cosine transform in CAL and C languages targeting the Epiphany architecture. Cal2Many framework is used for generating C code by using the CAL code. The results (number of source line of code and execution duration) of hand-written and generated implementations with different memory and communication configurations are compared to each other and presented in the paper. The final results of the paper shows that the generated code runs approximately 30% slower while having 78% less source line of code.

5.2 Parallel QR Decomposition on Epiphany Architecture - Paper II

This paper presents development and comparison of three different parallel approaches of QR decomposition written both in CAL and C languages and executed on Epiphany architecture. Additionally, an evaluation of the Cal2Many
code generation framework is performed by comparing CAL and C versions of the approaches.

QR decomposition has many useful applications such as replacing matrix inversions to avoid precision loss and reduce number of operations, being a part of the solution to the linear least squares problem and being the basis of an eigenvalue algorithm (the QR algorithm).

The approaches are implemented with dataflow programming model and a custom communication library is used for communication between processing cores. The implementations are tested with different matrix sizes and varying memory and communication configurations. The results show that different algorithms show different performances with different configurations. Additionally, source line of code and footprint size differ from algorithm to algorithm.

5.3 Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis- Paper III

Computations on floating-point numbers are performed in countless areas. However implementing efficient floating-point arithmetics in hardware is a challenging task. Among the operations such as addition, subtraction, multiplication, and division, the latter is clearly the most challenging. It usually requires more area and more clock cycles.

In this paper, we implement a novel floating-point division hardware. This hardware performs an inversion on the divisor and then multiplies the result with the dividend. The inversion method is a combination of Parabolic Synthesis and Second Degree Interpolation [44, 43]. This hardware performs division on IEEE-754 single-precision floating-point numbers [47].

Pipelined and non-pipelined versions of the hardware is synthesized on Xilinx Ultrascale FPGA and compared to several other implementations which adapt different methods. The results show that the proposed implementation needs less resources than the comparable implementations which leads to shorter delay. In case of throughput, the proposed implementation outperforms most of the other works except one implementation on an Altera Stratix-V FPGA. This is due to the difference in the size of the multipliers used in the DSP slices. Stratix-V provides larger multipliers which would change the results of the proposed implementation significantly.

The proposed implementation provides high performance with low resource usage. Therefore, it is suitable for high performance embedded systems executing intensive computations such as complex division and cubic interpolation which are presented in the next paper (Paper IV).
5.4 Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs - Paper IV

This paper presents the generic form of the design method that is given in chapter 4 and its initial realization. The initial realization covers the first 3 steps of the design method and generates single core architectures with accelerators. These 3 steps are (I) application development, (II) analysis and code generation, and (III) accelerator integration. The system integration step is partially covered by generating tiles consisting of memory, core and accelerator. The Cal2Many back-end that generates combination of C and Chisel is presented in this paper.

Two case studies are used to generate architectures and evaluate the design method and the automation tools. The first case study is the QR decomposition that is described in section 2.5.2. The second case study is the Autofocus criterion calculation that is presented in section 2.5.3.

The applications are analyzed and accelerators are generated for the identified hot-spots. In order to evaluate the efficiency of the hardware generation tool, the accelerators are manually developed as well. The performance measurements are done by executing the applications on the cycle accurate emulators generated for each architecture. The area usage results are gained by synthesizing the generated verilog code on Virtex Ultrascale FPGA via Xilinx tools. The performance and area usage of the accelerators are compared. Additionally, performance and area usage of architectures with and without the accelerators are measured and compared.

The results show that when the accelerator is generated automatically, its area usage, in best case does not increase and in the worst increases by 12%. The performance of the accelerator (in terms of number of clock cycles) does not decrease in the best case, however in the worst case it decreases by 9%. Further optimizations in the code generation may increase the performance and decrease the area usage. Finally, the cores with accelerators outperform the cores without accelerators by factors between 2.8 and 4.8×.

The initial realization of the design method shows that the architectures can be developed with a small effort thanks to the automation tools and the specialized architectures are more efficient than their general purpose counterparts.

5.5 Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit - Paper V

Square root is a fundamental operation in many areas and applications. However, developing a hardware block that performs this operation efficiently is a
significant challenge. It requires large amount of hardware resources and clock cycles.

In this paper, we develop a hardware block that performs square root operation based on a novel method that is a combination of parabolic synthesis and second degree interpolation developed by Hertz et al. [44, 43]. The hardware block supports the IEEE-754 single-precision floating-point number format [47].

The hardware block is developed with and without pipelining in Chisel language. Verilog code is generated from the Chisel code and synthesized on two different FPGAs which are Virtex-7 and Virtex Ultrascale. Performance, area usage and error behaviour of the method is measured and compared to other methods.

The developed hardware shows better performance than the other studies in terms of latency and uses fewer LUTs and FFs. Thus, it is suitable for high-performance embedded systems. In case of error distribution, the behaviour of our hardware is far better than the others. Our hardware produces an error distribution around zero which prevents accumulated error problems whereas the hardware that is based on the CORDIC method show an error distribution away from zero.

5.6 A Configurable Two Dimensional Mesh Network-on-Chip Implementation in Chisel - Paper VI

The design method proposed in paper IV targets generating manycore architectures. However, the processor generator tool (rocket chip generator) does not have a scalable network-on-chip. Therefore we implement a two dimensional mesh network-on-chip (NoC) router and a network interface in Chisel and connect this router to the (RISC-V) cores generated by the tools. The Chisel language is chosen due to the rocket chip generator, which is developed in Chisel. The router is implemented in VHDL as well.

This paper presents the description of the NoC router, related studies, hardware resource usage of different sized networks and performance analysis of a 4 × 4 network. The router has connections to the routers on four directions and to the local core through the network interface. It implements a deadlock free XY routing protocol for the network traffic and a Round Robin scheduler to receive packets from the other routers or the local core.

The hardware resource usage results are obtained by synthesizing different sizes of networks on a Xilinx Ultrascale FPGA via Xilinx tools. A router that is connected to four other routers and a network interface requires around 2.1k LUTs and 1.2FFs. The maximum clock rate varies between 250 MHz to 384 MHz for 16 × 16 and 1 × 2 networks respectively. The Chisel implementation is compared to the VHDL implementation. The source line of code for the Chisel
implementation is approximately 10% lower than the VHDL implementation. The hardware usage and clock rate results are similar.

The performance measurements of the Chisel implementation, which consists of latency and throughput, are performed via cycle accurate emulations with different synthetic traffic patterns including uniform random, bit complement and hot point patterns. The latency varies from 15 cycles to 95 cycles based on the injection rate of the network packets. The average throughput is around 1.3 packets/cycle. The performance decreases dramatically with the hot point pattern, which creates a significant congestion around a single router.

The results are competitive when compared to the state of the art. Especially, in terms of latency, our network outperforms the majority of the related works. The performance results can be increased further by increasing the data link sizes between the routers.

5.7 A Framework to Generate Domain-Specific Manycore Architectures from Dataflow Programs - Paper VII

The design method for developing heterogeneous manycore architectures was first introduced in paper IV, however, the realization was able to generate single core architectures. In this paper we complete the realization of the design method by integrating a two dimensional mesh network-on-chip to the tool chain and extending the code generation tool.

The default on-chip network that is supported by the rocket chip generator is crossbar which does not scale well. We developed a network-on-chip router in paper VI, that implements the two dimensional mesh topology, in Chisel and integrated it into the rocket chip generator to support scalability and generate architectures with many cores. Additionally, the code generator tool is extended to support core-to-core communication and generation of multiple cores with multiple accelerators.

The complete design method is used to generate several architectures for two different case studies from different domains. The first case study is the Autofocus criterion calculation that is used in paper IV as well. In this paper, different parallel (dual core and 13 core) versions are implemented. The second case study is the first convolution layer of GoogLeNet [85]. This application is executed on a single core as a proof of concept to show that the design method can be used for different application domains.

The performance measurements are done by executing the applications on the cycle accurate emulators generated by rocket chip generator for each architecture. The area usage results are gained by synthesizing the generated verilog code on a Virtex UltraScale FPGA via Xilinx tools. When compared to a single core, the autofocus criterion calculation is executed up to 20 times faster by a generated architecture that consists of 13 cores and accelerators. This
architecture shows similar performance to Epiphany architecture in terms of clock cycles per pixel. However, due to lower clock frequency, the Epiphany architecture shows a better performance in terms of throughput (pixels/second). The convolution is executed 4 times faster when an accelerator is generated and utilized.

The results show that with the design method it is possible to generate efficient architectures for different application domains. The performance of the architectures increase when the number of cores are increased. Additionally, usage of accelerator is another factor that increases the performance. In certain cases, usage of accelerators decreases the hardware resource usage due to avoiding the general purpose floating-point unit which is a large component.

This study shows that our framework can facilitate designing new manycore architectures and exploring different design options.
Chapter 6
Conclusions and Future Work

6.1 Conclusions

Processor architectures evolve continuously due to the technological advancements and the growing performance demand. We now live in the era of big data and artificial intelligence each of which require very high computation power. The current evolution direction for the processor architectures is towards domain-specific heterogeneous manycores due to power consumption and performance requirements, and heterogeneous structure of the applications in terms of tasks running in parallel. However, there is still a lack of knowledge on these architectures and there are challenges to be addressed before they can be adopted by mainstream.

In this thesis, we divide the challenges into software and hardware dimensions and address the challenges in both dimensions. The software challenges are the difficulty of programming these architectures efficiently and being able to execute the same implementation on different architectures (achieving code portability). The hardware challenges are the difficulty of developing such complex architectures with custom hardware that is tailored for the requirements of the target applications to achieve high-performance and better energy efficiency, and determining the design configurations.

In order to facilitate software development and increase code portability we develop new software development tools such as source to source compilers and propose utilization of these tools. We evaluate a dataflow language and dataflow programming model to reveal their impact on parallel programming. Additionally, we use a source-to-source code generation framework to facilitate parallel programming by increasing the abstraction level and hiding the details of target architectures. We evaluate this tool with several case studies and optimize it based on the evaluation results. We develop the case studies in the dataflow language, feed them to the framework and generate implementations in the native language of a commercial homogeneous manycore
architecture. We execute these implementations on the architecture, measure its performance and identify certain drawbacks.

Our first set of results show that usage of software development tools and domain-specific languages, which increase the abstraction level, lead to reduced development effort (25-50%) and increased software portability with a small penalty on the performance (2-17%). It additionally decreases the level of knowledge required about the target architectures. However, the tools usually cover only homogeneous architectures, thus there is still a need for progress in these tools, languages and programming models in order to exploit heterogeneity in manycores.

In order to address the challenges in the hardware dimension, we first define heterogeneity and its different forms in manycore architectures together with their advantages and disadvantages. Then we develop a design method to design manycore architectures with different configurations adopting a form of heterogeneity. The aim of the design method is to design application/domain-specific manycore architectures by augmenting the cores with application-specific custom hardware. In order to facilitate the design, we develop tools to automate the steps of the design method including a hardware library to generate application specific accelerators. The library includes hardware blocks, which perform division and square root operations based on a novel method. We evaluate the design method and the automation tools with case studies from different application domains. We use performance, hardware resource usage and maximum clock frequency measurements for the evaluation.

The second set of results show that an automated design method for developing manycore architecture increases the productivity of the architecture developers. Additionally, the results prove that parallelism and specialization increases the performance of the architecture. Automatically generated hardware accelerators perform as good as the manually written counterparts for some case studies and reaches 96% of the performance in another case study. The generated manycore architectures show a competitive performance. Furthermore, they reveal the impact of design configurations such as number of cores (parallelization) and acceleration usage (specialization). The hardware resource usage depends on the requirement of the applications. The accelerators may enable avoiding certain general purpose computation components to save resources. In other scenarios, the hardware resource usage increases due to the accelerators.

Finally, we conclude that software development tools facilitate programming manycore architectures with a small performance penalty, and automated design method facilitates developing manycore architectures and explore different design options. We believe that the future of manycore architectures will continue going towards heterogeneity and software/hardware development tools are necessary to make efficient use of these architectures and explore their design space to reveal possible configurations.
6.2 Future Work

We propose a design method and automate its major steps. However, there are still a few minor steps, which can be automated such as extending the RoCC interface to integrate the accelerators into the rocket chip generator, generating a configuration class and setting the configurations within the rocket chip generator, and creating the linker script and the start-up code based on the configurations. Moreover, the generator can be extended with different network-on-chip and core implementations. The NoC implementations can support new topologies or routing protocols whereas the core implementations can have different micro-architectures such as out of order execution or different number of pipeline stages.

When the 2D mesh NoC was integrated to the rocket chip generator, the default crossbar network was partially removed. Removing the entire crossbar network will decrease the resource usage and increase the maximum clock rates of the generated architectures. The current 2D mesh NoC implementation can be optimized to have larger data links between routers to increase the performance in terms of latency and throughput. This may increase the hardware usage, therefore there is a need for a study to find a sweet spot for this trade-off.

The hardware library can be extended to generate accelerators with different capabilities. There can even be a library for accelerators. This is possible by developing new applications. It is not evaluated yet, however, one can combine the tiles generated for different applications to support efficient execution of a number of applications on the same manycore architecture. We have measured the performance of the architectures with the cycle accurate emulators. In order to avoid the slow emulations, one can try measuring the performance on the actual hardware (FPGA), however, this requires a significant amount of work.

There is a room for improvement in code generation, especially on the hardware side. The communication mechanism that is generated supports channel buffer size of 1 token (usually an int or float). Using a dynamic buffer size, based on the requirements of the CAL application may avoid potential deadlocks. Loops are not supported while generating hardware, this support can be added to the hardware generation.

With all the proposed future works, the efficiency of the tools and the generated architectures will increase. Furthermore, the contribution of the automated design method to the design space exploration of new manycore architectures will grow.
References


[22] Rocket Core. Available online [https://github.com/ucb-bar/rocket], 2017. (Cited on page 22.)


REFERENCES


REFERENCES


[60] Shaoshan Liu, Won W Ro, Chen Liu, Alfredo C Salas, Christophe Cérin, and Jean-Luc Gaudiot. Eha: The extremely heterogeneous architecture.
REFERENCES


Appendix A

Paper I
An evaluation of code generation of dataflow languages on manycore architectures
Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström, and Mingkun Yang

An Evaluation of Code Generation of Dataflow Languages on Manycore Architectures

Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström and Mingkun Yang
Centre for Research on Embedded Systems Halmstad University, Halmstad, Sweden

{suleyman.savas, essayas.gebrewahid, zain-ul-abdin, tomas.nordstrom}@hh.se, minyan09@student.hh.se

Abstract—Today computer architectures are shifting from single core to manycores due to several reasons such as performance demands, power and heat limitations. However, shifting to manycores results in additional complexities, especially with regard to efficient development of applications. Hence there is a need to raise the abstraction level of development techniques for the manycores while exposing the inherent parallelism in the applications. One promising class of programming languages is dataflow languages and in this paper we evaluate and optimize the code generation for one such language, CAL. We have also developed a communication library to support the inter-core communication. The code generation can target multiple architectures, but the results presented in this paper is focused on Adapteva’s many core architecture Epiphany. We use the two-dimensional inverse discrete cosine transform (2D-IDCT) as our benchmark and compare our code generation from CAL with a hand-written implementation developed in C. Several optimizations in the code generation as well as in the communication library are described, and we have observed that the most critical optimization is reducing the number of external memory accesses. Combining all optimizations we have been able to reduce the difference in execution time between auto-generated and hand-written implementations from a factor of 4.3x down to a factor of only 1.3x.

Index Terms—Manycore, Dataflow Languages, code generation, Actor Machine, 2D-IDCT, Epiphany, evaluation

I. INTRODUCTION

In the last decade we have seen a transition from single core processors to multicore and manycore architectures, we furthermore see a transition to a distributed memory model where each core contains part of the common memory. These transitions have emphasized the need for programming languages that better express inherent parallelism available in the targeted applications. As a result, new languages based on parallel and concurrent programming paradigms have emerged [1], and the dataflow programming model [2], [3], [4] seems to be a particularly good candidate for the big class of streaming applications where data is processed as a continuous stream. There is however a lack of compilation frameworks, supporting dataflow languages, that generate code for the latest generation of manycore architectures.

In this paper we will evaluate and optimize a compilation framework targeting Adapteva’s manycore architecture Epiphany [5]. This architecture is an example of the next generation of manycore architectures that have been developed in order to address the power and thermal dissipation limitations of traditional single core processors. In the Epiphany architecture the memory is distributed among all cores and a shared address space allows all cores to directly access data located in other cores. However, in contrast to traditional multicore there is no cache memory and no direct hardware support for memory coherence.

The dataflow language used in this work is the CAL actor language (CAL) [4], [6], which is a modern dataflow language that is adopted by the MPEG Reconfigurable Video Coding (RVC) [7] working group as part of their standardization efforts. In [8] the CAL compilation framework evaluated and optimized in this paper is described. Our code generator uses a simple machine model, called actor machine [9], to model actors in a dataflow application. We have also introduced an action execution intermediate representation [8] to support program portability and we currently support generating sequential C, parallel (manycore) C, and aJava/aStruct languages. These intermediate representations are further introduced in section IV. For the parallel C backend the code generator utilizes an in-house developed communication library [10] to support the communication between actors.

In order to evaluate various optimization methods for our CAL code generator we will use a CAL implementation of the Two-Dimensional Inverse Discrete Cosine Transform (2D-IDCT) as a case study. We will compare the performance of our automatically generated code (from CAL) with a hand-written implementation programmed in C. From this comparison a number of optimization opportunities has been found and we will be able to evaluate the benefit of the various optimizations implemented.

II. RELATED WORKS

CAL compilers have already been targeting a variety of platforms, including single-core processors, multicore processors, and programmable hardware. The Cal2C compiler [11] targets the single-core processors by generating sequential C-code. The Open-RVC CAL Compiler (ORCC) [12] generates multithreaded C-code that can execute on a multicore processor using dedicated run-time system libraries. Similarly the d2c [13] compiler produces C-code that makes use of POSIX threads to execute on multicore processors.
In this paper, we present and evaluate a code generator for manycore architectures that use actor machine [9] and action execution intermediate representations (IRs) together with a backend for the target architecture and a custom communication library. Executing a CAL actor includes choosing an action that satisfy the required conditions and firing the action. In other CAL code generators [11] [12], all required conditions of the actions are tested at once, thus common conditions among actions will be tested more than once. However in our work, the schedule generated by the Actor machine stores the values of the conditions and avoids a second test on the conditions that are common.

In order to increase the portability of our compilation tool we have used an imperative IR called action execution intermediate representation (AEIR) which distinguishes computation and communication parts of the CAL applications. Hence it becomes easier to port the application to another architecture just by replacing the communication part with architecture specific communication operations. In addition, our compilation tool generates separate code for each actor instance so that it could be executed on individual processing cores. Therefore we do not require any run-time system support for concurrent execution of actors, in contrast to ORCC and d2c.

III. BACKGROUND

A. Epiphany Architecture

Adapteva’s manycore architecture Epiphany [5] is a two-dimensional array of processing cores connected by a mesh network-on-chip. Each core has a floating-point RISC CPU, a direct memory access (DMA) engine, memory banks and a network interface for communication between processing cores. An overview of the Epiphany architecture can be seen in Figure 1.

Fig. 1. Epiphany architecture overview.

In the Epiphany architecture each core is a superscalar, floating-point, RISC CPU that can execute two floating point operations and a 64-bit memory load operation on every clock cycle. The cores are organized in a 2D mesh topology with only nearest-neighbor connections. Each core contains a network interface, a multi-channel DMA engine, a multicore address decoder, and a network-monitor. The on-chip node-to-node communication latencies are 1.5 clock cycles per routing hop, with zero startup overhead. The network consists of three parallel networks which are used individually for writing on-chip, writing off-chip, and all read requests, respectively. Due to the differences between the networks, writes are approximately 16 times faster than reads for on-chip transactions. The transactions are done by using dimension-order routing (X-Y routing), which means that the data first travels along the row and then along the column. The DMA engine is able to generate a double-word transaction on every clock cycle and has its own dedicated 64-bit port to the local memory. The Epiphany architecture uses a distributed memory model with a single, flat address space. Each core has its own aliased, local memory range which has a size of 32kB. The local memory of each core is accessible globally from any other core by using the globally addressable IDs. However, even though all the internal memory of each core is mapped to the global address space, the cost (latency) of accessing them is not uniform as it depends on the number of hops and contention in the mesh network.

B. Dataflow Programming

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations or actors that operate on the dataflow. Dataflow programming emphasizes the movement of data and models programs as a series of connected actors. Each actor only communicates through explicitly defined input and output connectors and functions like a black box. As soon as all of its inputs become valid, an actor runs asynchronous from all other actors. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems. Dataflow programming is especially suited for streaming applications, where data is processed as a continuous stream, as, e.g., video, radar, or base station signal processing.

A number of dataflow languages and techniques were studied [2], [3], and one of the interesting modern dataflow languages is CAL Actor Language which has recently been used in the standardization of MPEG Reconfigurable Video Coding (RVC).

C. CAL Actor Language

A CAL dataflow program consists of stateful operators, called actors, that transform input streams of data objects (tokens) into output streams. The actors are connected by FIFO channels and they consist of code blocks called actions. These actions transform the input data into output data, usually with the state of the actor changed.

An application consists of a network of actors. When executed, an actor consumes the tokens on the input ports and produces new tokens on the output ports. Following example shows an actor which has one input port, two output ports and two actions.

```plaintext
actor Split() int I1 ==> P, N:
  action I1:[a] ==> P:[a]
  guard a >= 0 end
  action I1:[a] ==> N:[a]
  guard a < 0 end
```
The first line declares the actor name, followed by a list of parameters (which is empty, in this case) and the declaration of the input (I) and output ports (P and N). The second line defines an action [14] and the third line defines a guard statement. In many cases there are additional conditions which need to be satisfied for an action to fire. These conditions can be specified using the guard statements, as seen in the Split actor. The code schedule in CAL permits to determine the transitions between the different states. Each state transition consists of three parts: the original state, a list of action tags, and the following state. For instance, end

\begin{verbatim}
init (readT) --> waitA;

init is the original state, readT is an action tag and waitA is the following state. In this example, if readT action is executed within the init state, the actor will go to the next state which is waitA.

CAL has a network language (NL) in order to configure the structure of the application. The structure is built by instantiating the actors, defining channels and connecting the actors via these channels. Following example demonstrates a network with one input port and one output port. In the network, one processor is instantiated and it is connected to the input and output ports of the network.

\begin{verbatim}
network idct X0 ==> Y0
entities
  scale = Scale()
structure
  X0 --> scale.X0
  scale.Y0 --> Y0
end
\end{verbatim}

IV. CODE GENERATION

The code generator starts from high-level CAL implementation and generates a native implementation. Figure 2 shows the CAL compilation framework. In CAL, actors execute by firing of actions that satisfy all the required conditions like availability of tokens, value of the tokens and internal state of the actor. In code generation, the code generator has to consider the ordering of these conditions and the number of the tests performed before an action is fired. To schedule the testing of firing conditions of an action we have used a simple actor model called actor machine (AM) [9]. In addition, we also have used an Action Execution Intermediate Representation (AEIR) that bring us closer to imperative languages [8]. As shown in Figure 2, in our compilation process each CAL actor is translated to an AM that is then translated to AEIR. Finally, the AEIR is used by three backends that generate target specific code. Currently we have three backends: a uniprocessor backend that generates sequential C code for a general purpose processor, an Epiphany backend that generates parallel C code for Epiphany, and Ambric Backend that generates a Java and a Struct for Ambric massively-parallel processor array [15].

A. Intermediate Representation

1) Actor Machine: AM is a controller that provide an explicit order to test all the firing conditions of all actions in an actor. AM is made up of a set of states that memorize the values of all conditions in an actor. Each state has a set of AM instructions that leads to a destination states once applied on the state. These AM instructions can be

- a test to perform a test on a guard or input port for availability of token,
- an exec for an execution of an action that satisfy all the firing conditions, or
- a wait to change information about absence of tokens to unknown, so that a test on an input port can be performed after a while.

Cedersjö and Janneck [16] have presented and examined several ways of transforming an actor to an actor machine. The code generator used in this paper translates CAL actors to actor machines which memorize all the conditions and have at most one AM instruction per state.

2) Action Execution Intermediate Representation: AEIR is a data structure that is applicable to generate code for imperative languages like C, C++ and java. It has constructs for expressions, statements, function declarations and function calls. Translation of AM to AEIR deals with two main tasks. The first task is the translation of CAL constructs to imperative constructs. This includes CAL actions, variable declarations, functions, statements and expressions. The second task is the implementation of the AM that is the translation of the AM to a sequential action scheduler. The states and the instructions of the AM are translated to programing language constructs: the states are translated to unique labels or if-statement on state variable, test nodes to if-statements, exec to function call and wait to a call for a target specific operation that pause the execution of a process. The action scheduler is kept as a separate imperative function in the AEIR. In a prior work [8], we have presented the use of AEIR to generate sequential and parallel C code. Section V-B present three different parallel C codes generated for Epiphany architecture.

B. Mapping CAL actors

There is no well-defined semantics for NL to guide an implementation. In our work we have used Kahn Process Networks (KPN) [17] for Ambric and extended Dataflow Process Networks (DPN) [18] to generate sequential C code for a general purpose CPU and parallel C code for Epiphany. Ambrics proprietary tool comes with communication API and mapping and scheduling protocols that support KPN. Thus, for Ambric we have adapted the code generation in accordance with the existing support for KPN. Ambric code generation generate a Java object and a Struct code for each actor and top level design file for the application. The aJava source code is compiled into executable code by using Ambrics compiler.

For DPN we have implemented a communication API that use bounded buffers to connect output ports to input ports:
these are blocking for writing when the buffer is full, but allows peeking without blocking. If there are multiple actors on a single core, writing on full buffer blocks the running of the current actor and gives control to the other actors. The flattened network is used to generate a round-robin scheduler for the sequential C code and a host code for Epiphany that map the actors on separate processors.

1) The Dataflow Communication Library: All the communication that is done between the actors is done through FIFO buffers, thus making this functionality a key component for the compilation of the applications developed in CAL onto a manycore architecture. We suggest to implement this functionality as a dataflow communication library, with only five basic functions. In addition to the traditional functions of write and read from the FIFO buffer we have added functions such as connect which logically connects two actors, disconnect which logically disconnects the actors, and finally a function end_of_transmission that flushes the buffer and indicates that there are no further tokens to be sent.

When implementing these buffers on the Epiphany architecture, two special features of this architecture need to be considered. First one is the speed difference between read and write transactions (as mentioned earlier, writes are faster). The second one is the potential use of DMA to speed up memory transfer and allowing the processor to do processing in parallel to the memory transfer.

We have investigated three ways to implement the FIFO buffering. The first implementation is a ‘one-end-buffer’ which places the buffer inside the input port in the destination core. (Putting the buffer on the source core would result in reading from a remote core instead of writing to a remote core and thus a tenfold slowdown.) The communication overhead resides completely in the sender.

If we want to use DMA, we need to have a buffer on both the sender and receiver side. In the second implementation (‘two-end-buffer’) each core performs read and write transactions on its local memory and then uses DMA to transfer the data. This transfer is performed when both sides are ready, which requires that the sender’s buffer is full and the receiver’s buffer is empty. Even though we are using DMA for data transfer, the processor will be busy waiting for the DMA to finish. This is obviously not very efficient, and this method should be seen as a transition to our third method.

To allow the DMA to work in parallel with the processing core, we have implemented a ‘double-two-end-buffer’ method, which introduces two “ping-pong” buffers on each side of the communication channel. This allows the cores to work on one local buffer while the data from the other buffer is transferred to the other core by means of DMA.

If the token production rate on the sending actor is equivalent to the token consumption rate on the receiving actor, it is expected that the ‘double-two-end-buffer’ method should be the most efficient. On the other hand, if there is a big imbalance in the production/consumption rate, all three buffering methods will suffer from blocking, after the buffer gets full.

We have implemented the broadcast capability in all three implementations of the communication API. This mechanism is used when the output channel of an actor needs to be connected to multiple input channels of the other actors. Hence, the same output data can be written to multiple input ports and actors. In all implementations, the synchronization between sender and receiver is achieved by using flags belonging to the buffers. These flags indicate the availability of the data or the empty space in the buffers. These flags are kept in the local memories of the cores due to being polled continuously in a busy waiting loop, during the read and write operations.

The write and read function calls are designed to be asynchronous (non-blocking calls) by default, however, they will be blocking if the buffer is full or empty respectively. The functions end_of_transmission, connect, and disconnect calls will always be blocking. More details of the communication library are described in a different work [19].

V. IMPLEMENTATION

As our case study, we use the two-dimensional inverse discrete cosine transform (2D-IDCT), which is a component of MPEG video decoders. The CAL implementation of the 2D-IDCT is used as input to the code generator and as a reference implementation for the hand-written code. This implementation consists of 15 actors and these actors are mapped one-on-one to the Epiphany architecture using 15 out of 16 available cores. This implementation of the 2D-IDCT uses two one-dimensional inverse discrete cosine transforms after each
other, with all actors connected in a pipeline fashion. One dimensional IDCTs are highlighted with the dotted rectangles in Figure 3.

Both hand-written and automatically generated implementations map the actors onto the cores by using the serpentine layout which can be seen in Figure 4. This layout takes into account the physical layout of the cores on the chip and the routing method of the architecture (X-Y routing in this case), so that consecutive actors are mapped into neighboring cores.

Fig. 4. Serpentine layout to map 2D-IDCT to cores on the chip. RowSort is mapped onto core 0, Clip is mapped onto the core13 whereas core 12 is not used.

A. Hand-written Implementation

The hand-written implementation of 2D-IDCT is based on the reference CAL code. The implementation includes an embedded communication mechanism. This communication mechanism is similar to the ‘one-end-buffer’ implementation of the communication library presented in the previous section. However, the buffer size is 1 which means an actor can send only one data token at a time to a target actor. Due to the write operation being cheaper than the read operation on the NoC of the Epiphany architecture, the write function of the communication mechanism writes the data to the memory of the remote core and the read function of this mechanism reads the data from the local memory of the core. The read and the write operations are both blocking. The communicating cores use the flags of the ports in order to inform each other about the availability of data elements or empty space in the buffer. Two actors had input availability control for all input channels in the guard states. Therefore, these actors have been modified further to be able to combine multiple communication channels into 1 channel. There is no broadcasting functionality. Majority of the actors have 1/1 input/output ratio however there are two actors which have 1/2 input/output ratio and two actors which have 2/1 input/output ratio. This information is used in order to run each actor in a loop until they produce the expected number of outputs.

B. Automatically Generated Implementations

The results from the initial and three different optimizations of our compilation framework can be found in Table I. The initial implementation, which is called as ‘Uninlined’ in Table I, is intended to be structurally similar to the original CAL actors. Actions are translated to two functions: action_body for the body of the action and a separate function, action_guard, to evaluate the conditions of the guard of the action. For proper initialization the variable declaration of the actions are pushed for both functions. In the action scheduler AMs test instruction are translated to function call to the action_guard and test_input functions, and exec instructions are translated to a call for action_body.

The second implementation, called as ‘Inlined G’ in Table I, optimizes the initial one by using a pass that inline all action_guard functions. Beside function inlining, the pass also analyzes the scope of the variables and remove unused variable initializations.

The third implementation, ‘Inlined G & B’ in Table I, inlines both action_guard and action_body function calls. Like the second pass, third pass also use variable scope information
to eliminate unnecessary variable initializations. Thus, instead of copying the entire action variable initializations, only those variables which are used by the guard or the body of the action are initialized.

All generated implementations use the custom communication library. The parameters of the communication library such as buffer sizes, communication mechanism version and mapping layout are controlled by the code generator however for this evaluation they are kept fixed at the best configuration found after exploring the design space.

C. Differences Between Hand-written and Generated Implementations

To be able to understand the reasons behind the performance differences and evaluate the code generation, we can note some important differences between the automatically generated and the hand-written implementations. These differences are:

- State machines, in the generated code, often include high number of branches and states.
- Actions are generated as separate functions whereas in the hand-written code, they are combined in one function. Function calls add overhead (due to dealing with the stack) and decrease the performance.
- Different communication mechanisms are used. The mechanism, that is used in the generated code, is meant to be generic, e.g. broadcasting is supported which is not necessary for the particular application used as the case study. Hence the generated code includes communication mechanism overhead when compared to the hand-written code.
- Several further optimizations are implemented in the hand-written code, such as reduction of channel numbers. In the CAL implementation, actors are connected to each other with 4 channels. These channels are combined into one channel in the hand-written code in order to decrease network setup overhead and simplify the usage of channels.

VI. RESULTS AND DISCUSSION

We tested both of the implementations on the Epiphany III microchip running at 600 MHz. Among the different mechanisms of the communication library, we got the best results with the ‘one-end-buffer’ mechanism. Hence, we use the results of this mechanism for the discussions and the evaluation. In the test cases, we read the input data elements from the external memory and write the output elements back to the same external memory.

As can be seen in the last row of Table I the execution of the hand-written implementation takes 0.14 ms when the input size is 64 data elements (a block) and 93.6 ms when the input size is 64k data elements. These are the results which we aim to achieve with the automatically generated implementation.

Initially, the execution of the auto-generated implementation took around 1.2 ms for 64 data elements and around 405 ms for the 64k data elements. These results are given in the first row of Table I. The hand-written implementation out-performed the auto-generated implementation by 8.8x for 64 data elements and by 4.3x for 64k data elements. After further analysis, it is realized that the main bottleneck for both implementations is the external memory access. Most of the execution time is spent on reading the data from the external memory. In addition to the reading, writing to the external memory consumes some time as well. In the hand-written implementation, the external memory access for each input and output data is at a minimal level. However, in the auto-generated implementation, there were more external memory accesses per input and output elements due to both the communication library and the code generation.

In the auto-generated implementation, the variables, which were residing in the external memory, are moved to the local memory of the cores. This significantly decreased the execution time from 405 ms to 154 ms for 64k data elements, which resulted in a throughput increase by 63%. The performance of the auto-generated implementation got as close as 1.6x to the performance of the hand-written implementation. This optimization is referred as memory optimization in Table I.

![Fig. 5. Total execution times for the auto-generated implementation together with the times spent for the computation.](image-url)
correspondence of the guard conditions. The results, when this optimization is applied, can be seen in the third (‘Mem opt + Inlined G’) row of the Table I. As the second optimization on the code generator, we inlined the functions which corresponded to the action bodies. This optimization provided us a new auto-generated implementation and the results of this implementation corresponds to the fourth (‘Mem opt + Inlined G & B’) row of the Table I. By applying these optimizations to the code generation, we decreased the execution time from 154 ms to 125 ms for 64k data elements. This decrease in the execution time corresponds to 18% throughput increase. Using this result, the hand-written implementation shows 1.3x better performance. The difference is reduced further from 1.6x to 1.3x.

After the optimizations on the communication library, apart from the input and output data elements, no data element is kept in the external memory. There are a few data elements which reside in the shared memory and used for synchronization between the host and core0, however, they can be ignored since they are used only once before the cores start running. Even if the only access to the external memory is the access to input and output data elements we find that this external access is dominating the execution time. Figure 5 shows the total execution times of individual cores together with the time spent for the computation. The difference between the total time and the actual computation time gives the time that is spent on the communication and waiting for I/O operations due to the slow external memory accesses. In Figure 5 we can see that for most of the cores around 90% of this time (the difference of the total and the computation times) is spent on reading the input data elements from the external memory. The external memory accesses are performed by only the first and last cores. For 64k elements, the first core performs 64k external memory reads whereas the last core performs 1k external memory reads and 64k external memory writes. In the hand-written implementation, the first core runs for approximately 56 million cycles and 51.9 million of these cycles are spent for reading the input from the external memory. The situation is more or less the same in the auto-generated implementation. Due to the slow read operations in the first core, the other cores wait for the input data and this waiting time is the main contributor of the difference between the total and the computation times. The cost of the write operations, in the last core, which is far less than the cost of the read operations, are mostly hidden by the cost of the read operations. The cost of the communication between the cores and the computation times are mostly hidden by the cost of the external memory accesses. In summary, in this application, the cores are mostly blocked while reading the input data elements and they are blocked for a shorter time while writing the output data elements.

An additional aspect for comparison between the hand-written and the auto-generated implementation is the code size. In Table II the machine code size of each actor/core is given for both hand-written and auto-generated implementations. (The auto-generated implementation, which includes memory optimization and inlined action guards & action bodies, is used.) These code sizes are obtained by running the ‘size’ command in linux command line. The .elf files, which are produced by the Epiphany linker, are used as input parameters to the ‘size’ command. The auto-generated implementation has 71% more machine code when compared to the hand-written implementation.

Another interesting aspect is the difference in development effort writing the 2D-IDCT application in C and writing it in CAL. In Table III the number of source lines of code (SLOC) for each actor for the hand-written (native - C) implementation and the reference CAL implementation is compared. We see that in total, 495 SLOC are needed to implement the 2D-IDCT application in CAL while more than four times as many (2229 SLOC) is needed for the C implementation. This clearly indicates the expressiveness of the CAL language and the usefulness of the CAL compilation tool as described in this paper.

### VII. Future Work

For further evaluation of the code generation and the custom communication library, a more complex application such as the entire MPEG-4 decoder implementation can be both automatically generated and manually written. This work might need combining and splitting the actors. Hence the compilation framework may need to be extended. In addition to the evaluation of the code generation, the mapping approach proposed by Mirza et al. [20] can be tested with this application. The performance of both hand-written and auto-generated implementations might be increased by further optimizations. We observed that the main bottleneck for both of the implementations is the read operations performed on the external memory. An optimization could be using the write operations

### Table II

<table>
<thead>
<tr>
<th>Cores / Actors</th>
<th>Hand-written</th>
<th>Auto-generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>8,636</td>
<td>8,077</td>
</tr>
<tr>
<td>core0 / RowSort</td>
<td>5,488</td>
<td>9,020</td>
</tr>
<tr>
<td>core1 / Scale</td>
<td>5,556</td>
<td>9,568</td>
</tr>
<tr>
<td>core2 / Combine</td>
<td>5,012</td>
<td>9,372</td>
</tr>
<tr>
<td>core3 / ShuffleFly</td>
<td>5,080</td>
<td>8,888</td>
</tr>
<tr>
<td>core4 / Shuffle</td>
<td>5,556</td>
<td>9,452</td>
</tr>
<tr>
<td>core5 / Final</td>
<td>4,952</td>
<td>9,060</td>
</tr>
<tr>
<td>core6 / Transpose</td>
<td>5,708</td>
<td>10,628</td>
</tr>
<tr>
<td>core7 / Scale</td>
<td>5,244</td>
<td>9,568</td>
</tr>
<tr>
<td>core8 / Combine</td>
<td>5,008</td>
<td>9,376</td>
</tr>
<tr>
<td>core9 / ShuffleFly</td>
<td>4,920</td>
<td>8,892</td>
</tr>
<tr>
<td>core10 / Shuffle</td>
<td>5,080</td>
<td>9,456</td>
</tr>
<tr>
<td>core11 / Final</td>
<td>4,952</td>
<td>9,064</td>
</tr>
<tr>
<td>core12 / Shift</td>
<td>4,952</td>
<td>9,044</td>
</tr>
<tr>
<td>core13 / Retranspose</td>
<td>7,160</td>
<td>10,564</td>
</tr>
<tr>
<td>core14 / Clip</td>
<td>5,044</td>
<td>10,236</td>
</tr>
<tr>
<td>Total</td>
<td>87,712</td>
<td>150,265</td>
</tr>
</tbody>
</table>
instead of read operations. Such as, instead of the Epiphany core(s) reading the input data from the external memory, the host can write the data to the Epiphany core’s memory. The synchronization between the host and the Epiphany core can be obtained by using the blocking write operations. Hence no read operations would be used. Another optimization for the communication library would be customization of the functionality. E.g the operations such as broadcasting can be turned on and off for different applications in order not to add overhead when the operation is not needed. A radical optimization can be replacing the communication mechanism with a more efficient one which can make better use of the specific features of the architecture.

In our test application, each actor is connected only to the neighbor actors. Hence the communication is not very complicated, and choosing the best fitting layout is not a complex task. However, with larger applications when the communication patterns get more complicated, choosing the mapping layout, quickly becomes a very complex task. For e.g individual actors might be connected to several other actors. Hence the mapping method would need to take network usage, communication frequency between actors etc. into account while searching for the best fitting layout. Mirza et al. [20] propose a solution to the mapping, path selection and router configuration problems. They refer that their approach is capable of exploring a range of solutions while letting the designer to adjust the importance of various design parameters.

As an extension to the compilation framework we used in our work, new backends can be implemented and integrated to the compilation tools in order to target other manycore architectures and the code generation can be tested on these architectures.

VIII. Conclusions

Manycore architectures are emerging to meet the performance demands of high-performance embedded applications and overcome the power dissipation constraints of the existing technologies. A similar trend is visible in programming languages in the form of dataflow languages that are naturally suitable for streaming applications. This paper deals with the evaluation of a compilation framework along with a custom communication library that takes CAL actor language (CAL) code as input and generates parallel C code targeting Epiphany manycore architecture. As a case study we have used a CAL implementation of a 2D-IDCT application and compare the automatically generated code from the proposed tool-chain with a hand-optimized native-C implementation.

The preliminary results reveal that the hand-written implementation has 4.3x better throughput performance with respect to the auto-generated implementation. After memory access optimizations on the communication library, the auto-generated implementation gained 63% throughput increase. With further optimizations on the code generation, the throughput of the auto-generated implementation was further improved by 18%. To sum up, we are able to decrease the difference in execution time between the hand-written and the auto-generated implementations from a factor of 4.3x to 1.3x. In terms of the development effort by considering source lines of code metric, we observe that the CAL based approach requires 4.5x less lines of source code when compared to the hand-written implementation.

To conclude we are able to achieve competitive results of execution time with respect to the hand-written implementation and the use of high-level language approach leads to reduced development effort. We also foresee that our compilation methodology will result in focusing on optimizing the tool-chain to produce efficient implementations rather than manual optimizations performed on each application.

ACKNOWLEDGEMENTS

The authors would like to thank Adapteva Inc. for giving access to their software development suite and hardware board. This research is part of the CERES research program funded by the Knowledge Foundation and HiPEC project funded by Swedish Foundation for Strategic Research (SSF).

REFERENCES


Appendix B

Paper II
Dataflow Implementation of QR Decomposition on a Manycore

Süleyman Savas, Sebastian Raase, Essayas Gebrewahid, Zain Ul-Abdin, and Tomas Nordström

Dataflow Implementation of QR Decomposition on a Manycore

Süleyman Savas, Sebastian Raase, Essayas Gebrewahid, Zain Ul-Abdin and Tomas Nordström
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Suleyman.Savas@hh.se, Sebastian.Raase@hh.se, Essayas.Gebrewahid@hh.se, Zain-Ul-Abdin@hh.se, Tomas.Nordstrom@hh.se

ABSTRACT
While parallel computer architectures have become mainstream, application development on them is still challenging. There is a need for new tools, languages and programming models. Additionally, there is a lack of knowledge about the performance of parallel approaches of basic but important operations, such as the QR decomposition of a matrix, on current commercial manycore architectures.

This paper evaluates a high level dataflow language (CAL), a source-to-source compiler (Cal2Many) and three QR decomposition algorithms (Givens Rotations, Householder and Gram-Schmidt). The algorithms are implemented both in CAL and hand-optimized C languages, executed on Adapteva’s Epiphany manycore architecture and evaluated with respect to performance, scalability and development effort.

The performance of the CAL (generated C) implementations gets as good as 25% slower than the hand-written versions. They require an average of 25% fewer lines of source code without significantly increasing the binary size. Development effort is reduced and debugging is significantly simplified. The implementations executed on Epiphany cores outperform the GNU scientific library on the host ARM processor of the Parallella board by up to 30x.

1. INTRODUCTION
Computer architectures are moving towards manycores for reasons such as performance and energy efficiency. The required parallelism to use these architectures efficiently requires new software tools, languages and programming models to abstract the differences between different architectures away. This reduces required knowledge about the architectures and their specific programming language extensions. Even with tools, writing efficient parallel applications is challenging and there is a lack of knowledge on performance of common applications such as QR decomposition when executed on manycores.

QR decomposition (QRD) [8] is one of the major factorizations in linear algebra. It is well known to be numerically stable and has many useful applications such as replacing matrix inversions to avoid precision loss and reduce the number of operations, being a part of the solution to the linear least squares problem and being the basis of an eigenvalue algorithm (the QR algorithm).

In this paper, we evaluate Cal2Many [6] source-to-source compiler, which translates CAL [5] code to native code for multiple manycore architectures. As a case study, we implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) both in CAL and in native C for Adapteva’s Epiphany [11] architecture. All implementations use our own communications library [13]. We used the Parallella platform to evaluate our implementations in terms of performance, development effort and scalability.

2. BACKGROUND

2.1 CAL Actor Language
The CAL actor language is a dataflow language consisting of actors and channels. Actors are stateful operators which execute code blocks (actions), take inputs and produce outputs usually with changing the state of the actor. The channels are used to connect the actors to each other. Therefore, interaction among actors happens only via input and output ports. CAL actors take a step by ‘firing’ actions that satisfy all the required conditions. These conditions depend on the value and the number of input tokens, and on the actor’s internal state. The actors are instantiated and connected to each other via Network Language (NL) included in CAL.

2.2 Cal2Many
The Cal2Many compilation framework contains two intermediate representations (IRs): Actor Machines (AM) [10] and Action Execution IR (AEIR) [6]. Each actor is first translated to an AM, which describes how to schedule execution of actions. To execute AM, its constructs have to be transformed to a different programming language constructs, which have different implementations in different programming languages on different platforms. To stay language-agnostic and get closer to a sequential action scheduler, AEIR is introduced. Epiphany backend generates C code using our custom communications library and generates channels and mapping of actor instances by using the NL.

2.3 QR Decomposition
QR decomposition is decomposition of a matrix into an upper triangular matrix $R$ and an orthogonal matrix $Q$. The
equation of a QRD for a square matrix \( A \) is simply \( A = QR \).
The matrix \( A \) does not necessarily need to be square. The equation for an \( m \times n \) matrix, where \( m \geq n \), is as follows:
\[
A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = [Q_1 \quad Q_2] \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1.
\]

We have implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) in both CAL and native C for the Epiphany architecture.

2.4 Epiphany Architecture

Adapteva’s manycore architecture is a two-dimensional array of cores connected by a mesh network-on-chip [11]. It operates in a shared, flat 32-bit address space. The network-on-chip is made of three meshes called \( rMesh \), \( cMesh \) and \( xMesh \). The former is used exclusively for read requests, while the latter two carry write transactions destined for on-chip and off-chip, respectively. The mesh uses a static XY routing algorithm.

Each core contains a single-precision floating-point RISC CPU, 32KB of local memory, a two-channel DMA engine and a mesh interface. Two event timers allow cycle accurate measurements of different events.

We have used the Parallella-16 platform, which contains a 16-core Epiphany-III running at 600 MHz in addition to a dual-core ARM Cortex-A9 host processor running at 667 MHz.

2.5 Communication Library

We have implemented a custom communications library for the Epiphany architecture [13], which is used in all implementations. It is centered around token-based, unidirectional communication channels. These channels support blocking read, write and non-blocking peek operations and allow checking the current number of immediately read- or writable tokens for block-free operations.

A global table describes all channels in the system. Local data structures and buffers are allocated at run-time by each core. Channels are implemented as ring buffers and it is possible to use the event timers to gauge blocking overhead.

3. RELATED WORKS

There are many approaches for QRD, however, they focus either on architectures or scheduling and use only one algorithm. We have executed three algorithms in two programming languages and compared them to each other.

Buttari et al. [3] present a QRD method where they run a sequence of small tasks on square blocks of data and use a dynamic scheduling mechanism to assign tasks to computational units. They execute small tasks in a dataflow fashion.

Hadri et al. [9] present a QRD method for shared-memory multicore architectures and modified an existing algorithm to perform panel factorization in parallel. They aim tall or small square matrices whereas we aim square matrices.

Agullo et al. [2] implement a three step QRD on a multicore CPU which is enhanced with multiple GPU cores. They divide the decomposition into a sequence of tasks as we have done. Then they schedule these tasks on to individual computational units. Their approach suffers if the number of CPUs and GPUs are not the same.

While the CAL2C compiler [15] generates sequential C code, the Open-RVC CAL Compiler (ORCC) [12] and d2c [4] compilers generate multi-threaded C code, but require dedicated run-time system libraries. Our compilation framework generates separate C code for each actor instance to be executed on individual cores and does not require any run-time system support.

4. IMPLEMENTATIONS

All implementations use the same library to implement communication between the Epiphany cores, which is used by the Cal2Many as well. However, the communication characteristics differ naturally between algorithms and, to a lesser extent, between the hand-written (C) and the generated (CAL) implementations of the same algorithm.

4.1 Givens Rotations

The Givens Rotations (GR) algorithm applies a set of unitary rotation \( G \) matrices to the data matrix \( A \). In each step, one of the sub-diagonal values of the matrix \( A \) is turned into zero, forming the \( R \) matrix. The multiplication of all rotation matrices forms the orthogonal \( Q \) matrix.

We implemented Givens Rotations (GR) with a modified Gentleman-Kung systolic array [7] [16] using 1, 5 and 12 cores individually. In parallel versions, two cores are used for distributing and collecting inputs and outputs, the rest of the cores are used for computations.

The implementations consist of 4 types of units named as cells. Boundary and inner cells perform the computation while splitter and joiner cells distribute and collect data. Figure 1 gives the layout of the systolic array mapped on the \( 4 \times 4 \) core matrix of the Epiphany architecture. The inputs are read row by row. Each row is divided into four pieces and distributed to the first row of \( 4 \times 4 \) Epiphany core matrix. Token size is defined as the size of each piece and the cores send and receive one token at a time for communication. Apart from the input elements, \( c, s \) and final \( r \) values are communicated.

Each cell calculates one \( r \) value. For \( 4 \times 4 \) matrix, each cell is mapped onto an Epiphany core. However for \( 16 \times 16 \) matrix, 16 boundary and 120 inner cells are required. Therefore the implementations are modified to combine a number of cells in one single core. Each core has to store the calculated \( r \) values which becomes a problem when large matrices are used. For \( 512 \times 512 \) input matrix, an inner core consists of \( 128 \times 128 \) inner cells which results in storing a \( 128 \times 128 \) \( r \) matrix. Since \( r \) is a float, the required memory is \( 4 \times 128 \times 128 = 65536 \) bytes. Due to the local memory limitation, the largest matrix size that can be decomposed with Givens Rotations method is \( 256 \times 256 \). The implementations can scale when the number of cores increases, i.e., with 64 cores the implementations can decompose a \( 1024 \times 1024 \) matrix.

4.2 Householder Transformation

The Householder (HHH) algorithm describes a reflection of a vector across a hyperplane containing the origin [18].

In our implementation, the Epiphany cores are connected as a one-dimensional chain of processing elements. Each core handles an equal amount of matrix columns and runs the same program. The communication is wave-like and next-neighbor only.

First, the input matrix is shifted into the processing chain, column by column, until it is fully distributed among the cores. Then, the last core in the chain computes a reflection vector \( w \) for each of its columns, updates them, and sends the vector towards the beginning of the chain, forming a
communication wave each. All previous cores forward these vectors and update their own columns. After these waves have been forwarded by the penultimate core, it – after updating its own columns – computes its own reflection vectors and sends them along, forming new waves. When a core has sent its last reflection vector, it will form a final set of waves containing its result columns.

4.3 Gram-Schmidt

The Gram-Schmidt (GS) algorithm produces the upper-triangular matrix $R$ row-by-row and the orthogonal matrix $Q$ as a set of column vectors $q$ from the columns of the data matrix $A$ in a sequence of steps. In each step, we pick a column $a$ of the matrix $A$. The dot product of this column with itself is calculated. Then, the square root of the result is taken to generate an element of matrix $R$. This element is later used to normalize the column $a$ to produce a column of matrix $Q$. Then the column of matrix $A$ is updated by subtracting a multiple of vector $q$ with a value from matrix $R$ to produce an orthogonalized vector that is then used to compute the next columns of matrix $R$.

Both CAL and C implementations work as a chain of processes that work on a certain number of columns, depending on the number of processes and the matrix size. All processes perform the steps required to compute a column of matrix $R$. In the first step, each process stores its local columns and pushes the remaining columns into the chain. In the second step, the processes read and forward the previous orthogonalized vectors after they use the vectors to produce the elements of matrix $R$ and to update their local columns. In the third step, the processes compute and forward the local, orthogonalized vectors using their updated local columns. In the final step, the processes forward the elements of matrix $R$. The implementation can scale up to any number of processors $num_p$ and any $m \times n$ matrix, where $n$ is a multiple of $num_p$.

5. RESULTS & DISCUSSION

Our implementations are executed on the Parallella board and tested with different number of cores and different sized square matrices.

5.1 Performance Analysis

There are several aspects such as memory usage, input size and number of cores which affect the performances. Therefore we executed the implementations with different configurations and Table 1 presents the number of execution cycles while using external and internal memories to store input and output matrices. Additionally, last two columns give the number of cores used and the total code size in bytes.

As a reference point, we measured the execution time of a QRD implementation, that is a part of GNU Scientific Library [1], on the ARM processor of Parallella board. The library implementation uses Householder approach with double precision, whereas the Epiphany architecture supports only single precision. Decomposition of a $128 \times 128$ matrix on a single ARM core takes 90 milliseconds whereas on the Epiphany cores it takes 6.9, 4.1 and 2.9 milliseconds (shown in Table 1) for hand-written parallel GR, GS and HH implementations respectively. GR implementation is the only one that can decompose $256 \times 256$ matrices. While decomposing $128 \times 128$ matrices, it outperforms the library by 13x, however, with $256 \times 256$ matrices this number increases to 23x. When the matrix sizes increase, parallel implementations perform better due to increased computation/communication ratio as a result of communication patterns used in the implementations.

We implemented a message passing mechanism for inter-core communication as a library. If messages (or as we call them, ‘tokens’) are passed very frequently, the overhead of communication dominates the execution time due to library calls and copy operations. The communication patterns we use in our implementations keep the number of message passes constant regardless the matrix size. If the matrix sizes increase, instead of increasing the number of messages, the size is increased. By keeping the number constant, we avoid extra overhead of calling library functions. The communication cost still increases due to increased size of data that needs to be copied. However, it does not increase as fast as the cost of computation. Hence its effect on overall execution time decreases.

Figure 2 shows performance of each hand-written algorithm decomposing a $64 \times 64$ input matrix that is stored in the internal memory. We chose this matrix size because
it is the biggest size that is supported by all implementations with different core numbers (except for the single core implementation of Gram-Schmidt algorithm due to memory limitations). Gram-Schmidt implementation can decompose 64 × 64 matrix by using 2, 4, 8 and 16 cores and achieve 4x speed-up by going from 2 core to 16 cores. Householder implementation can decompose on 1, 2, 4, 8 and 16 cores and achieve 5.2x speed-up going from single core to 16 cores. Givens Rotations implementation decomposes the same matrix on 1, 3 and 10 computational cores due to its structure and achieves 3.4x speed-up while going from single core to 10 cores. When decomposing small sized matrices such as 64 × 64, communication overhead plays a significant role in the execution time and decreases the speed-up. However, as the matrix size increases, the effect of communication decreases.

Table 1: Execution times (in clock cycles and milliseconds), source lines of code, number of used cores and code size in bytes, 128 × 128 matrix. GR = Givens Rotations, HH = Householder, GS = Gram-Schmidt

<table>
<thead>
<tr>
<th></th>
<th>External mem</th>
<th>Local mem</th>
<th>SLoC</th>
<th>#cores</th>
<th>Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR Hand-written</td>
<td>9.51 M 15.8 ms</td>
<td>4.16 M 6.9 ms</td>
<td>647</td>
<td>12</td>
<td>71 k</td>
</tr>
<tr>
<td>GR CAL</td>
<td>10.45 M 17.4 ms</td>
<td>6.11 M 10.1 ms</td>
<td>400</td>
<td>12</td>
<td>72 k</td>
</tr>
<tr>
<td>HH Hand-written</td>
<td>10.45 M 17.4 ms</td>
<td>1.76 M 2.9 ms</td>
<td>219</td>
<td>16</td>
<td>225 k</td>
</tr>
<tr>
<td>HH CAL</td>
<td>10.69 M 17.8 ms</td>
<td>2.00 M 3.3 ms</td>
<td>170</td>
<td>16</td>
<td>223 k</td>
</tr>
<tr>
<td>GS Hand-written</td>
<td>11.17 M 18.6 ms</td>
<td>2.47 M 4.1 ms</td>
<td>188</td>
<td>16</td>
<td>179 k</td>
</tr>
<tr>
<td>GS CAL</td>
<td>11.40 M 19.0 ms</td>
<td>2.70 M 4.5 ms</td>
<td>160</td>
<td>16</td>
<td>193 k</td>
</tr>
</tbody>
</table>

Figure 2: Execution cycles for GS, HH and GR respectively with different number of cores and 64x64 input matrix. X axis represents number of cores and Y axis represents number of clock cycles.

In our previous works [17, 14] we have experienced that the external memory access can be a bottleneck in the Epiphany architecture due to the slow link between the memory and the processing cores. Therefore we tested the implementations with and without using the external memory for storing the input matrix. We observed that when the input size increases, which means increased computation, the influence of external memory decreases however, it is still a bottleneck. Table 1 shows that using external memory to store the input matrix slows down the execution by 56% to 83% depending on the implementation. Givens Rotations seems to be the least influenced algorithm due to overlap between memory reads and computation.

An interesting point is the comparison of C and CAL implementations. Looking at Table 1, one can see that there is not much difference between hand-written and CAL implementations while using external memory. Even while using internal memory, the difference increases only for the GR implementation due to a slightly more complicated structure compared to the other implementations such as different communication pattern or having different number of channels depending on the position of the core. These small details increase the number of actions which are converted into functions in C. Overhead of repetitive calls to these functions increases the execution time. The main reasons of slow down for the generated code is having a scheduler and function calls. In the hand-written code, the actions are combined in the main function and there is no scheduler. Therefore there is neither function call overheads nor switch statements to arrange the order of these functions.

5.2 Productivity Analysis

In addition to performance, we compare the CAL and C implementations in terms of development effort, which is difficult to measure. It should be noted that our approaches to QR have been implemented by three developers, who have more knowledge and experience with C rather than CAL. Nonetheless, the CAL implementations required about 25% less source code, while the binary code size stays approximately the same. The numbers shown in Table 1 do not include the additional source code for the ARM host or the communication library, since it is similar in all cases and used by the Cal2Many code generator as well.

In each case, about half of the development time was spent on understanding both the problem and the algorithm, which is independent of the choice of programming language. The actual implementation times for each algorithm varied. One of the developers had no prior experience with CAL and required approximately 20% more time for the CAL version, while the other developers required slightly more time for their C implementations. While this is by no means a hard measure, it provides an idea on the complexity of CAL. More importantly, the CAL implementations are completely oblivious of the underlying hardware and are easily portable, while the C implementations are quite restricted to the Epiphany system architecture. This higher level of abstraction also reduced the debugging effort, which is extremely cumbersome in low-level parallel programming.
6. CONCLUSIONS

Parallel implementations show up to 30x better performance in terms of execution time when compared to the library implementation. However, the Givens Rotations implementation shows that with bigger matrices the speed-up increases. Since the implementations are scalable, we believe that with larger local memory or larger number of cores the implementations can decompose bigger matrices and achieve higher speed-ups.

While using the external memory, Givens Rotations is slightly better than the other implementations, however, Householder method outperforms the others when local memory is used. Givens Rotations has higher amount of computation and more overlap between memory reads and computation. High computation amount increases the execution time when local memory is used. However, when external memory is used, due to the overlap, it shows the best performance. One should keep in mind that the number of cores is smaller for Givens Rotations implementation. In case of development complexity, Table 1 shows that implementing Givens Rotation requires more coding whereas the other implementations have similar code sizes.

As an average result of the three implementations, the generated code runs 4.5% slower than the hand-written code while using the external memory. When the internal memory is used, the slowdown is around 17% whereas the average source lines of code that is needed for the CAL implementations is 25% smaller. When the required knowledge level and development time and complexity is taken into account, the slowdown seems reasonable. It is easier to develop and debug parallel applications in CAL rather than in low level languages provided by manycore developers. The custom communication library reduces the burden however, it does not help with debugging and requires debugging itself.

In the future, Parallella board with 64 cores can be used for gaining higher speed-ups, and in order to decrease the effect of communication even further, direct memory access feature of Epiphany architecture can be analyzed.

Acknowledgment

This work was supported in part by the ESCHER Project funded by the Swedish Knowledge Foundation, the HiPECC project funded by the Swedish Foundation for Strategic Research, and the ELLIIT Strategic Research Initiative funded by the Swedish Government.

7. REFERENCES


Appendix C

Paper III
Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis

Süleyman Savas, Erik Hertz, Tomas Nordström, and Zain Ul-Abdin

IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2017), Bochum, Germany, July 3-5, 2017.
Efficient Single-Precision Floating-Point Division
Using Harmonized Parabolic Synthesis

Süleyman Savas, Erik Hertz, Tomas Nordström, Zain Ul-Abdin
School of Information Technology, Halmstad University, Halmstad, Sweden

{suleyman.savas, erik.hertz, tomas.nordstrom, zain-ul-abdin}@hh.se

Abstract—This paper proposes a novel method for performing division on floating-point numbers represented in IEEE-754 single-precision (binary32) format. The method is based on an inverter, implemented as a combination of Parabolic Synthesis and second-degree interpolation, followed by a multiplier. It is implemented with and without pipeline stages individually and synthesized while targeting a Xilinx Ultrascale FPGA.

The implementations show better resource usage and latency results when compared to other implementations based on different methods. In case of throughput, the proposed method outperforms most of the other works, however, some Altera FPGAs achieve higher clock rate due to the differences in the DSP slice multiplier design.

Due to the small size, low latency and high throughput, the presented floating-point division unit is suitable for high performance embedded systems and can be integrated into accelerators or be used as a stand-alone accelerator.

I. INTRODUCTION

Floating-point division is a crucial arithmetic operation required by a vast number of applications including signal processing. For instance, the advanced image creating sensors in synthetic aperture radar systems perform complex calculations on huge sets of data in real-time and these calculations include interpolation and correlation calculations, which consist of significant amount of floating-point operations [1], [2]. However, it is quite challenging to implement efficient floating-point arithmetics in hardware. Division itself is the most challenging basic operation to implement among the others such as addition, multiplication and subtraction. It requires larger area and usually achieves relatively lower performance.

In this paper, we implement a single-precision division method that performs division on floating-point numbers represented in IEEE-754 standard [3]. The implementations are synthesized and executed on a field-programmable gate array (FPGA) for validation and evaluation.

The proposed method consists of two main steps. The first step is the inversion of the divisor and the second step is the multiplication of inverted divisor and the dividend. In the inversion step Harmonized Parabolic Synthesis is used as the approximation method, which is a combination of Parabolic Synthesis and Second-Degree Interpolation [4], [5]. When compared to other methods, Parabolic Synthesis methodology converges faster and entails faster computation and smaller chip area, which in turn leads to lower power consumption. In the Harmonized Parabolic Synthesis methodology the Second-Degree Interpolation achieves the required accuracy by using intervals. The accuracy increases with the number of intervals and for single-precision floating-point inversion 64 intervals are sufficient.

The division hardware that is implemented in this paper is a part of an accelerator that performs cubic interpolation on single-precision floating-point numbers. The accelerator is implemented in Chisel language [6] to be integrated to a RISC-V [7] core via Rocketchip [8] system on chip generator.

The accelerator and the division hardware will be used as basic blocks for building domain-specific heterogeneous manycore architectures. Based on the requirement of applications, these custom blocks or other similar blocks will be integrated to simple cores. Many of these cores together with many custom blocks will form efficient heterogeneous manycore architectures targeting specific application domains. If an application does not use all of the cores with custom blocks, they can be shut-down by utilizing dark-silicon concept [9].

The rest of this paper is structured as follows: In section II we provide background knowledge on single-precision floating-point representation and the floating-point division algorithm that we have used. Section III discusses related work. Section IV presents the approach that we have used for implementing our algorithm. The implementation on an FPGA board is explained in details in Section V. Section VI provides the results which are compared to related work in section VII. Section VIII finalizes the paper with conclusions and future work.

II. BACKGROUND

In this section, we introduce the FPGA, floating-point number representation and the method used for performing division.

A. FPGA Features

The target platform in this paper is Virtex UltraScale XCVU095-2FFVA2104E FPGA from Xilinx. This FPGA is developed with 20 nm technology and consists of 537600 look-up tables (LUTs), 1075200 flip-flops (FFs), 1728 block RAMs and 768 DSP slices. Block rams are dual port with the size of 36 Kb and can be configured as dual 18 Kb blocks. The DSP slices include $27 \times 18$ two’s complement multiplier.
B. Binary32 - Single-precision floating-point numbers

Even though there are many possible floating-point number formats, the leading format by far is the IEEE Standard for Floating-Point Arithmetic (IEEE-754) [3]. This standard defines the representation of floating-point numbers and the arithmetic operations. In this paper, we will focus on binary32 numbers, which are more commonly known as single-precision floating-point numbers.

The IEEE 754 standard specifies a binary32 (single-precision floating-point) number as having:
- Sign bit $s$: 1 bit
- Exponent width $e$: 8 bits
- Significand (mantissa) $m$: 24 bits (23 explicitly stored)

as illustrated in Figure 1.

![Fig. 1. Single-Precision (binary32) floating-point representation in IEEE-754 format](image)

The sign bit determines the sign of the number. The exponent is an 8-bit signed integer and biased by $+127$. The true significand consists of 23 visible bits to the right of the decimal-point and 1 invisible leading bit to the left of decimal-point which is 1 unless the exponent is 0. The real value is calculated with the following formula:

$$(-1)^s \times (1 + m) \times 2^{e-127} \quad (1)$$

The exponent ranges from $-126$ to 127 because $-127$ (all zeros) and 128 (all ones) are reserved and indicate special cases. This format has the range of $\pm 3.4 \cdot 10^{38}$.

C. Floating-point division

An overview of division algorithms can be found in [10]. According to the author’s taxonomy, division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. These algorithms will differ in overall latency and area requirements. The algorithm we use, is a table look-up algorithm with an auxiliary function for decreasing the table size.

We implement the division $R = X/Y$ as an inversion of $Y$ ($T = 1/Y$) followed by a multiplication of $X$ ($R = X \cdot T$), as shown in Figure 2.

The floating-point multiplier [11] uses on-board DSP slices to multiply the significands and adder slices (carry logics) to add the exponents. The inverter utilizes the efficient Harmonized Parabolic Synthesis method [4], [5]. The Parabolic Synthesis method is founded on a multiplicative synthesis of factors, each of which is a second-order function. The more factors that are used, the higher is the accuracy of the approximation.

![Fig. 2. Overview of the division method. X = dividend, Y = divisor](image)

III. RELATED WORKS

There has been extensive research on implementing floating-point operations in hardware. The research has been focused on three aspects namely performance, area, and error characteristic. In the rest of this section, we first present approximation methods for calculating the inversion and then present the prior works about implementing division with single-precision.

There are several methodologies such as CORDIC [12], [13], Newton-Raphson [14], [15] and polynomial approximation [16], [17] for calculating the inverse of a number. However, these methods are additive, whereas the method used in this paper, Parabolic Synthesis, is multiplicative. This means that the approximation converges faster, which results in faster and smaller implementation.

With the introduction of the Parabolic Synthesis methodology, the following improvements were accomplished compared to CORDIC. First, due to a highly parallel architecture, a significant reduction of the propagation delay was achieved, which also leads to a significant reduction of the power consumption. Second, the Parabolic Synthesis methodology allows full control of the characteristics and distribution of the error, which opens an opportunity to use shorter word lengths and thereby gain area, speed and power.

Singh and Sasamal [18] implement single-precision division based on Newton-Raphson algorithm using subtractor and multiplier, which is designed using Vedic multiplication technique [19]. They use a Spartan 6 FPGA and require quite high amount of hardware resources when compared to the proposed method. Leeser and Wang [20] implement floating-point division with variable precision on a Xilinx Virtex-II FPGA. The division is based on look-up tables and taylor series expansion by Hung et al. [21], which uses a 12.5KB look-up table and two multiplications. Regardless of the FPGA, the memory requirement is more than the proposed implementations. It is difficult to compare the resource utilization as the underlying architecture is different between Virtex-II and Ultrascale series, however, they seem to use more
resources than the proposed implementations.

Prashanth et al. [22], Pasca [23], and Detrey and De Dinechin [24] implement different floating-point division algorithms on Altera FPGAs. Even though it is very difficult to do a comparison between these works and the proposed work in terms of the utilization and timing results, one can still see the clear difference in Table I.

Prashanth et al. [22] design a single-precision floating-point ALU including a non-restoring division block consisting of shifters, adders, and subtractors. In contrast to proposed implementations, they do not use DSPs or block RAMs and have quite high cycle count requirement. Pasca [23] presents both single-precision and double precision division architectures based on Newton-Raphson and piece-wise polynomial approximation methods. His work focuses on correct rounding, which comes with a cost of several extra operations whereas rounding has no cost in the proposed implementations. Additionally, memory requirement is more than the requirement of the proposed implementations. Detrey and De Dinechin [24] compute the mantissa as the quotient of the mantissas of the two operands by using a radix 4 SRT algorithm [25]. The exponent is the difference between the two exponents plus the bias. Their work does not utilize any block RAMs or DSPs, however, the requirements for the other resources are significantly higher than the method implemented in this paper.

The methodology, proposed in this paper, is based on an inversion block and a floating-point multiplication block. The novelty lies in the inversion block, which uses the Harmonized Parabolic Synthesis method. This block uses three look-up tables, 4 integer multipliers and 3 integer adders, all with different word lengths. The look-up tables consist of 64 words with 27, 17 and 12 bits of word length respectively. These are significantly smaller when compared to other table look-up methods for the same accuracy.

Care should be taken when comparing solutions implemented using different FPGA platforms, however, without access to the other methods a truly fair comparison has not been possible. Still we believe our solution to be a significant improvement to the solutions so far suggested in the literature.

IV. METHODOLOGY

The proposed method consists of two main steps as presented in Figure 2. The first step is the inversion of the divisor and the second step is the multiplication of the inverted divisor and the dividend. Both divisor and dividend are in IEEE-754 single-precision floating-point (binary32) format.

In the first step, as seen in Figure 3, the sign bit, the exponent bits, and the significand bits of the divisor are extracted. The significand bits are forwarded to the inverter where the Harmonized Parabolic Synthesis (approximation) is performed. In this block, the coefficients of the synthesis method are stored in look-up tables.

In parallel to the inversion, sign of the exponent is inverted while taking the bias into account as follows:

removing the bias:

\[ e' = e - 127 \]  \hfill (2)

inverting the sign and adding the bias:

\[ e'' = 254 - e \]  \hfill (4)

The exponent is further decreased by 1 unless the result of the inverter is 0.

\[ \text{If } m \neq 0 \{ \text{inversion block} \} \]

\[ \text{If } e = e - 1 \]

Sign bit, exponent with inverted sign, and the inverted significand are combined to form a floating-point number. This number is fed as the first input to a floating-point multiplier, which performs the second step of our method. The dividend is forwarded directly to the multiplier as the second input (shown in Figure 2).

In the multiplier, the significands are multiplied and adjusted, the exponents are added together and adjusted based on the resulting significand. The multiplication result of the significands is rounded with different rounding methods such as ‘round toward zero’, ‘round to +∞’, and ‘round to nearest (ties to even)’. ‘Round to zero’ provides the smallest max error and best error distribution when compared to other methods.

The inverter has been tested for every possible input. The max error is 1.18358967 × 10^{-7} (≈ 2^{-23.01}) which is smaller than the machine epsilon (ε) (upper bound for the error) that is commonly defined as 2^{-23} (by ISO C, Matlab, etc) for the single-precision floating-point format.

The division result, which is the output of the floating-point multiplication block, is tested for 2^{23}(8,388,608) inputs. The rounding errors for the tested inputs are smaller than 1 ULP.
\[ ULP(x) = \epsilon \times 2^{e'} \]  

where \( x \) is the result in the single-precision floating-point format, \( \epsilon \) is machine epsilon and \( e' \) is the (unbiased) exponent of \( x \).

V. IMPLEMENTATION DETAILS

The division hardware is implemented in Chisel language and Verilog code is generated. In the generated code, the coefficients were produced via constant wires and multiplexers. The code is modified to reside the coefficients in a block ram. Synthesis is performed by Xilinx tools. The target platform is Xilinx Virtex UltraScale XCVU095-2FFV12104E FPGA.

Two different implementations of the floating-point division hardware are presented in this paper. The first implementation consists of a single stage, whereas in the second implementation, the inverter is divided into 4 stages and the floating-point multiplier is divided into 2 stages to increase the clock frequency. Consequently the dividend, sign, and exponent bits of the divisor are delayed for four cycles before being fed to the floating-point multiplier. When the inverter and the multiplier are combined the total number of stages in the second implementation becomes 6.

The stages and other details of the hardware implementation of the first step are given in Figure 4. As shown in this figure, two 8-bit subtractors, a single 23-bit equation logic and a single multiplexer are used for computing the exponent of the inverted divisor. These components correspond to the flow of the exponent in Figure 3.

The significand bits of the divisor are connected to the input of the inverter. As shown in Figure 4, the inverter block consists of three look-up tables for storing the coefficients, four integer multipliers, two adders, and a single subtractor. Each look-up table stores 64 words, however, the length of the words change due to the changes in number of bits for storing the coefficients. The word lengths for \( l, j, \) and \( c \) coefficients are 27, 17, and 12, respectively. When synthesized, these look-up tables are stored in one block ram. The most significant 6 bits of the significand are used for addressing these look-up tables. The same address is fed to each table. Rest of the bits are fed to two multipliers one of which is actually a squarer.

Each integer multiplication unit in the inverter utilizes one DSP slice on the FPGA except the last unit, which requires two DSP slices due to having large inputs. The input and output sizes of the multipliers are \( 17 \times 12 \rightarrow 17, 17 \times 17 \rightarrow 18, 17 \times 17 \rightarrow 17, \) and \( 23 \times 25 \rightarrow 24 \). The subtractor and the two adders utilize 6, 6, and 7 carry logics, respectively.

The floating-point multiplier utilizes two DSP slices for multiplying the significands and an 8-bit adder to sum the exponents. Additionally, two 8-bit subtractors are used for adjusting the result exponent.

VI. RESULTS

Two different implementation of the division hardware with different number of stages have been evaluated. The implementations have different results in terms of clock frequency and resource usage. These results are presented in the first two rows of Table I.

The first implementation computes the division in one stage and takes 15.2 ns. Thus, the clock frequency becomes 65 MHz.
TABLE I
COMPARISON OF DIFFERENT IMPLEMENTATIONS OF SINGLE-PRECISION FLOATING-POINT DIVISION

<table>
<thead>
<tr>
<th>Published</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>Freq(MHz)</th>
<th>Period(ns)</th>
<th>Cycle</th>
<th>Latency(ns)</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed Imp 1</td>
<td>79</td>
<td>32</td>
<td>7</td>
<td>1</td>
<td>65</td>
<td>15.2</td>
<td>1</td>
<td>15.2</td>
<td>Xilinx Virtex Ultrascale</td>
</tr>
<tr>
<td>Proposed Imp 2</td>
<td>183</td>
<td>257</td>
<td>7</td>
<td>1</td>
<td>232</td>
<td>4.3</td>
<td>6</td>
<td>25.8</td>
<td>Xilinx Virtex Ultrascale</td>
</tr>
<tr>
<td>Singh2016[18]</td>
<td>10019</td>
<td>408</td>
<td>-</td>
<td>-</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Xilinx Spartan 6 SP605</td>
</tr>
<tr>
<td>Pasca2012[23]</td>
<td>426</td>
<td>408</td>
<td>4</td>
<td>2 M20K</td>
<td>400</td>
<td>2.5</td>
<td>15</td>
<td>37.5</td>
<td>Altera Stratix-V</td>
</tr>
<tr>
<td>Leeser2005[20]</td>
<td>335 slices x (2LUT, 2FF)</td>
<td>8</td>
<td>7</td>
<td>110</td>
<td>9</td>
<td>14</td>
<td>126</td>
<td>126</td>
<td>Xilinx Virtex-II</td>
</tr>
</tbody>
</table>

In the second implementation the inverter is divided into 4 stages whereas the floating point multiplier is divided into 2.

Stage4, that is shown in Figure 4, has the longest latency with 4.3 ns. The multiplier, in this stage, utilizes 2 DSP slices and causes a latency around 4 ns. The next stage consists of another multiplier utilizing 2 DSP slices for the multiplication of the mantissas and has a similar latency. Since it is not possible to divide these multiplications, the clock period is chosen as the latency of the Stage4 which is 4.3 ns and the stages are not divided further. As a result, the clock frequency is 232 MHz and the total latency is 4.3 × 6 = 25.8 ns.

The implementations use a pipelined approach and produce one result at each cycle. This means max throughput (number of floating point divisions per second) of each implementation is equal to its clock frequency.

Resource utilization of the whole implementation is given in Table I whereas Table II shows the utilization of each individual component for both implementations. The first implementation uses 32 registers to hold the result of the division. In the second implementation, the resource usage increases due to the stages. Some of the stage registers are moved out of the inverter block during optimization by Xilinx and they are included in the leaf cell registers in Table II. Most of these registers are used for delaying the dividend that needs to wait for the inverter result.

TABLE II
UTILIZATION RESULTS OF PROPOSED IMPLEMENTATIONS ON A XILINX VIRTEX ULTRASCALE FPGA

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Module</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp 1 (1 stage)</td>
<td>Inverter</td>
<td>40</td>
<td>-</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FPMult</td>
<td>36</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Leaf cells</td>
<td>3</td>
<td>32</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Imp 2 (6 stages)</td>
<td>Inverter</td>
<td>107</td>
<td>101</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FPMult</td>
<td>45</td>
<td>10</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Leaf cells</td>
<td>31</td>
<td>146</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Available</td>
<td>537600</td>
<td>1075200</td>
<td>768</td>
<td>1728</td>
<td></td>
</tr>
</tbody>
</table>

VII. DISCUSSION

Table I compares the utilization and timing results of our implementations with five prior works. The implementations can be compared based on two aspects; resource usage and timing. However, comparing implementations running on different FPGAs is very difficult, if not impossible. Hence we will mention only comparable results and try to give advantages and disadvantages of our implementations in comparison to the others.

The main advantage of our implementations is the low resource requirement and consequently small area usage. The implemented inverter design requires 3 look-up tables with 64 words in each, which sums up to 448 bytes and utilizes a single block RAM on the target FPGA. On the other hand, the method adopted by Leeser and Wang [20], which is another table look-up method, requires a look-up table with the size of 12.5 KB. Similarly, Pasca[23] uses look-up tables to store approximation results and targets an Altera FPGA. The implementation requires two M20K memory blocks, each of which consists of 12480 configurable bits. Another important resource is the DSP slices. Leeser and Wang [20] utilizes 8 DSP slices within a Virtex-II, which has 18 × 18 multipliers. The DSP utilization of the proposed implementations would remain as 7 with the same sized DSPs. The required number of slices is 4 for Pasca [23]. However, these DSP slices (on Altera FPGA) support 25 × 25 multiplication size, whereas the DSP slices on Virtex Ultrascale support 27 × 18. With larger sized DSP slices, the DSP requirement of the proposed implementations can be reduced to 5. Consequently, due to solving the bottlenecks in Stage4 and Stage5, the clock frequency can be increased significantly.

Within the implementations, which do not use look-up tables, the implementation of Prashanth [22] utilizes the least number of resources. However, it suffers from the long latency.

Comparing the timing results would hardly lead to any meaningful conclusion due to having different platforms for different implementations. Hence we present the timing results in Table I without going into any comparison to give a hint of the performance of our implementations on a high-end FPGA.

A brief study on increasing the accuracy shows that for every 4-bits accuracy, the number of intervals need to be doubled. The accuracy difference between double precision and single precision formats is 29 bits which would require doubling the number of intervals 8 times. This would increase the number of intervals from 64 to 16384 for each coefficient. The word length of the coefficients would increase as well however, it is difficult to estimate. We think that the word lengths would be at least doubled. This would increase the
memory by a factor of 8.

Current memory requirement is 448 bytes and for the double precision division, the memory requirement would probably be more than 917504 bytes (448 × 2³³). In addition to the memory size, the component sizes such as the sizes of adders and multipliers would increase as well due to the increase in the word length of the inputs and the coefficients. Still, we see from these calculations that the memory requirement would dominate the circuit design when implementing double-precision floating-point using our method.

VIII. CONCLUSION

This paper presents a novel method for single-precision floating-point division based on an inverter, which utilizes a Harmonized Parabolic Synthesis method, and a floating-point multiplier. The method is used in two different implementations. These implementations perform the divisions on floating-point numbers represented in IEEE-754 binary32 format. In the current state, exceptions and denormalized numbers are not taken into account. However, the error performance of the inverter is validated by exhaustive testing. The maximum error of the division is found to be under 1 ULP (unit in the last place) after testing with random inputs and some corner cases.

The two implementations differ in the number of pipeline stages used and consequently provide different results in terms of latency, throughput and resource usage. Low resource requirement and high throughput make this floating-point division unit suitable for high performance embedded systems. It can be integrated into an existing floating-point unit or used as a standalone accelerator.

Future work includes handling the exceptions and denormalized numbers, completing the cubic interpolation accelerator, which adapts the 6 stage division design, and integrating it to a RISC-V core. Additionally, we plan to implement some other basic operations such as square root and inverse square root with Parabolic Synthesis method, make them IEEE-754 compliant and use them as basic building blocks while building manycore architectures.

ACKNOWLEDGEMENT

This research is done in the HiPEC (High Performance Embedded Computing) and NGES (Towards Next Generation Embedded Systems: Utilizing Parallelism and Reconfigurability) projects, which are funded by Swedish Foundation for Strategic Research (SSF) and VINNOVA Strategic Innovation grant, respectively.

REFERENCES


Appendix D

Paper IV

Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs

Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström

Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs

Süleyman Savas *, Zain Ul-Abdin and Tomas Nordström

School of Information Technology, Halmstad University, Halmstad, 301 18, Sweden; zain-ul-abdin@hh.se (Z.U.-A.); Tomas.Nordstrom@hh.se (T.N.)

* Correspondence: suleyman.savas@hh.se; Tel.: +46-72-977-36-07

Received: 1 March 2018; Accepted: 21 April 2018; Published: 22 April 2018

Abstract: The last ten years have seen performance and power requirements pushing computer architectures using only a single core towards so-called manycore systems with hundreds of cores on a single chip. To further increase performance and energy efficiency, we are now seeing the development of heterogeneous architectures with specialized and accelerated cores. However, designing these heterogeneous systems is a challenging task due to their inherent complexity. We proposed an approach for designing domain-specific heterogeneous architectures based on instruction augmentation through the integration of hardware accelerators into simple cores. These hardware accelerators were determined based on their common use among applications within a certain domain. The objective was to generate heterogeneous architectures by integrating many of these accelerated cores and connecting them with a network-on-chip. The proposed approach aimed to ease the design of heterogeneous manycore architectures—and, consequently, exploration of the design space—by automating the design steps. To evaluate our approach, we enhanced our software tool chain with a tool that can generate accelerated cores from dataflow programs. This new tool chain was evaluated with the aid of two use cases: radar signal processing and mobile baseband processing. We could achieve an approximately 4 × improvement in performance, while executing complete applications on the augmented cores with a small impact (2.5–13%) on area usage. The generated accelerators are competitive, achieving more than 90% of the performance of hand-written implementations.

Keywords: heterogeneous architecture design; risc-v; dataflow; QR decomposition; domain-specific processor; accelerator; Autofocus; hardware software co-design

1. Introduction

Several highly demanding applications are currently in the process of being introduced, such as autonomous vehicles, 5G communication, and video surveillance and analytics. Many of these also include artificial intelligence and machine learning algorithms, which adds significantly to their computational demands. To support these applications on vehicles and other mobile devices, there is a need for embedded high-performance computing architectures to perform streaming computations in real time.

The present generation of multi/manycore architectures, along with general purpose graphics processing units (GPGPU), aim to address these computational demands by duplicating identical processing units or cores. Some companies are already pushing the number of cores on a chip as high as a thousand [1]. The first multi/manycores were produced by duplicating the processing cores, resulting in homogeneous architectures having several identical cores on the same die [2,3]. However, the streaming applications mentioned above are comprised of a large variety of tasks which are not necessarily identical. For example, a typical massive MIMO [4,5] application, which is the core of
5G communication technology, consists of a chain of tasks, including channel encoding/decoding, precoding, OFDM modulation, channel estimation, and MIMO detection, each of which performs different computations and requires different hardware resources for efficient performance and power consumption. Some tasks do not even perform any computation but consist of only memory operations such as shuffling or transposing a matrix. Techniques such as virtualization [6] and containerization [7] aim to execute heterogeneous tasks efficiently on processing systems by encapsulating them and minimizing runtime requirements. Additionally, technologies such as hyper-threading [8] provide simultaneous execution of threads on the same processor. However, the efficiency that can be achieved is still limited by the efficiency of the underlying hardware. To achieve the highest efficiency while executing the tasks on a manycore, it is necessary to optimize individual cores to the task at hand, thus introducing heterogeneity [9–12]. One core may be optimized for efficient fast Fourier transforms (FFT) used in OFDM modulation, whereas another core can be optimized for efficient QR decomposition for matrix inversion during precoding. However, designing heterogeneous architectures is a challenging task due to the complexity of these architectures. There can be many forms of heterogeneity based on the components and how they are inter-connected and used [12]. Therefore, finding the most suitable architecture for a target application requires design space exploration. This represents a further challenge, because of the diversity in the manycore design. In performing design space exploration, manycores are often simulated partially, or as a full system. Simulations allow hiding unnecessary architectural details when a certain component is tested and require no hardware development, which reduces costs. However, simulators are usually quite slow [13].

There are software tools to simulate manycore architectures, such as Gem5 [14], ZSim [15], Graphite [16], Sniper [17], and PriME [18], however, most of them do not support the kind of complete configurable system that allows configuration of parameters such as the types and numbers of processing units, memory units, custom hardware blocks, or the network-on-chip structure. There are full system simulators such as Gem5 and SimFlex [19], which provide detailed data on different components. However, the simulation time increases beyond a feasible limit as the system grows. Additionally, the simulators do not provide timing and area information, which can be obtained by taking a design to RTL or FPGA implementation. As a result, exploring the design space of heterogeneous manycore architectures through simulations becomes increasingly challenging. Further discussion on manycore simulation can be found in [13].

In this paper, we propose a design approach for high-performance, domain-specific heterogeneous architectures based on simple cores with application-specific extensions in the form of custom hardware blocks. With this approach, instead of finding an efficient architecture to run a certain application, we aim to build the architecture automatically with a tool-chain starting from dataflow programs. The design approach can be summarized as identifying the extensions within an application domain, integrating these extensions to simple cores and, (as a future goal) generating heterogeneous manycore architectures by connecting these cores with a network-on-chip (NoC). The paper covers identification and integration of the extensions with case studies and provide the insight of connecting the extended cores with a NoC. However, it does not cover the steps needed to generate a manycore architecture. When all the design steps are automated, our approach can be used for exploring the design space of heterogeneous manycore architectures. We define the extensions (custom hardware blocks) as the compute-intensive parts (hot-spots) of the applications within a domain such as radar, baseband, or audio/video signal processing. These blocks are integrated to simple cores by extending the instruction set. The cores are tasked to execute the control flow of the application and delegate the compute-intensive parts to these blocks, using them as accelerators. The cores with accelerator extensions will be referred to as “tiles” in the rest of the paper.

As the first step towards automating the design approach, we developed a software tool to automatically generate custom hardware blocks from dataflow applications. We extended our code generation framework, Cal2Many [20], which takes dataflow applications and generates target specific code for different architectures, by adding a back-end to generate a combination of C and Chisel [21]
code. We considered two case studies to evaluate the performance and area usage of the generated code. The first case study was QR decomposition (QRD), implemented using Givens Rotations method [22]. The QRD operation is used in numerous signal processing applications including massive MIMO [23]. The second case study was an autofocus criterion calculation application, which is a key component of synthetic aperture radar systems [24]. The chosen method performs cubic interpolation based on Neville’s algorithm [25] to calculating the autofocus criterion. The case studies are implemented in CAL dataflow programming language [26], which is a concurrent language with support for expressing parallelism. The compute-intensive parts of the case studies were identified through profiling and automatically generated as custom hardware (accelerators). The accelerators are integrated to a rocket core [27] that is based on RISC-V open-source instruction set architecture (ISA) [28]. Synthesizable designs for the integration of the rocket cores and the accelerators are generated via rocket chip system on chip generator [29]. The generated implementations were evaluated in terms of performance and area usage against hand-optimized implementations. The contributions made by this study can be summarized as:

- A generic method to design domain-specific heterogeneous manycore architectures with an emphasis on custom-designed tiles was proposed.
- An approach to design augmented cores (tiles) via instruction extension and hardware acceleration was realized, including development of a code generation tool to automate hardware accelerator generation directly from a dataflow application. This tool performs hardware/software codesign and generates C and Chisel code from CAL applications.
- The design method was evaluated using two case studies from baseband processing and radar signal processing. For these case studies, hand-written and automatically generated accelerators are used. The accelerators are integrated into a RISC-V core.

The remainder of the paper is structured as follows: Section 2 surveys the literature on related work. Section 3 describes the generic and realized versions of the proposed design approach. Section 4 introduces the case studies and provides details of how they are implemented. Section 5 presents the results of the case studies and a discussion of their results. Section 6 contains concluding remarks and some discussion of possible future works.

2. Related Works

In accordance with the focus of our work on generating heterogeneous manycore architectures, we provide here a review of related work on manycore design and custom hardware generation from high-level languages.

The first set of related works are on manycore generation. There are FPGA based manycore architectures developed by Sano et al. [30], Tanabe et al. [31] and Schurz et al. [32]. However, these studies develop a single architecture and do not propose any generic method for doing so. However, celerity [33], an accelerator-centric system-on-chip (SoC) design based on a combination of three tiers, does closely resemble our work. The first tier consists of five rocket cores capable of running Linux, whereas 506 smaller RISC-V cores reside in tier 2, with an accelerator in tier 3. The tiers are connected to each other with parallel links. The accelerator is generated using SystemC and high-level synthesis tools. In their design, the accelerator and the cores are placed in different tiers and all cores share the accelerator tier. In contrast, in our design, each core can have its own tightly-coupled accelerator, making the accelerator an instruction extension to the simple core. Additionally, our design starts from application development and uses application requirements to configure the architecture in terms of the number of cores, memory size, accelerator types, etc.

There are several tools available that generate hardware description from high-level languages. These tools support a variety of languages [34] including Lava [35] and Clash [36] which are functional languages similar to Chisel [21]. Clash and Lava are based on Haskell and they have compilers and support simulations. Additionally, Clash supports generating Verilog, VHDL and SystemVerilog
Chisel is based on Scala and has a compiler for simulation and generating Verilog code. However, the RISC-V tools that are used in this study require Chisel code for integrating accelerators into the rocket cores. Therefore, we have chosen Chisel as our high-level hardware description language. There are more common languages, which are C and C-like languages, that are used by the tools to generate a hardware description. Some of these tools are Xilinx Vivado [38], Calypto’s Catapult C [39], CoDeveloper from Impulse [40], eXcite from Y Explorations [41], Stratus from Cadence [42], and Symphony C from Synopsis [43]. In these tools, the developer is usually required to change their C-like code by adding pragmas and rewriting some code snippets to change the structure of the generated RTL design. Both Catapult [44] and CyberWorkBench [45] deal with generating controller and data paths. They require modifications to the C code and the data paths depend on the controller generation. In our design method, we do not generate any controller. The accelerators we generate consist of only data paths and control is implemented on the processing core. This difference can also be seen when comparing our work with that of Trajkovic et al. [46]. Trajkovic et al. [46] automatically generate processor cores from C code, including separate generation of the data path and the controller.

The Tensilica XPRES (Xtensa PRocessor Extension Synthesis) Compiler [47] automatically generates extensions, which are combinations of existing instructions. It uses a C/C++ compiler to analyze the application and find the candidate instructions. The instructions can be in the form of VLIW, vector, fused operations, or combinations of these. Therefore, the generated extensions are within the bounds of combinations of the existing instructions. Instruction extensions require modifications to the compiler and the decoder. The compiler is generated automatically. However, having a combination of instructions, including VLIW-style instructions, requires multiple parallel instruction decoders [47]. This increases the hardware cost, which may affect the clock frequency, and also limits the number of instructions that may potentially be executed in parallel [46].

Clark et al. [48] automated custom instruction generation using dataflow graphs. They discovered custom instruction candidates by exploring the dataflow graphs. They used a hardware library to estimate timing and area for the the combined primitive operations (subgraphs). Combination of a few subgraphs, to be executed on the same hardware, is performed to form a set of custom function units (CFUs). Performance and area are estimated for the set members and fed to a selection mechanism to choose the custom instruction. They also generalized the custom instruction to be present in the input code, which in turn requires changes to the compiler/simulator. However, with our method, the only change in the code is applied to the name of the action to be accelerated.

Koeplinger et al. [49] automatically generated accelerators for FPGAs from applications described in a high-level language using patterns, such as map, reduce, filter, and groupBy. Their framework performs high-level optimizations and describes hardware using their own intermediate representation which is based on a set of architectural templates. Their method requires the parallel patterns and pragmas to be present in the input code, which in turn requires changes to the compiler/simulator. However, with our method, the only change in the code is applied to the name of the action to be accelerated.

The SDSoC environment from Xilinx [50] only provides automatic generation of accelerators running on their Zynq and Ultrascale+ platforms, which significantly limits portability across different target platforms.

The CAL2HDL tool, developed by Janneck et al. [51], was the first attempt to generate hardware descriptions from CAL actor language. This tool transforms CAL application into an XML language-independent model before the OpenForge synthesizer is used to translate the application to a hardware description in Verilog. CAL2HDL supports a subset of CAL and the time for Verilog design generation increases dramatically with the design complexity, which limits its wider applicability. It generates a platform-dependent design that can be used only on Xilinx FPGAs. Another hardware description generator from CAL actor language is developed by Siret et al. [52]. Their tool generates VHDL, however, it lacks loop support. Bezati et al. [53] developed Xronos, aiming to support the ISO
subset of CAL actor language to generate RTL code. Xronos uses OpenForge and generates Verilog, similar to CAL2HDL, however the authors claimed that Xronos operates faster and the generated hardware uses fewer resources because of changes in the transformations applied to the IR that is used as the input to the OpenForge. In our work, we did not generate hardware to cover the whole application but only the tagged (hot-spot) actions. Therefore, our back-end does not perform all of the transformations that are covered by the other CAL tools. Additionally, we used different IRs, and generated hybrid code (C+Chisel) that embeds custom instructions (in assembly) into the C code to communicate with the generated hardware blocks.

The generated heterogeneous architecture does not require an explicit mapping/resource-discovery process to map the application onto corresponding components (core or accelerator) as our code generation framework already takes care of this by generating the code to fire the accelerator wherever necessary. However, if many of these extended cores (core + accelerator) are to be used in the same architecture, there is a need for a mapping approach such as HARD (Hybrid adaptive resource discovery) [54]. This approach might be somewhat too advanced due to its functionality support. The only significant information for a mapping approach would be the capability of the accelerator and the hot-spots of the applications. Hence, a simpler approach can be applied to map the hot-spots to the accelerators.

To summarize, we present a generic methodology to design domain-specific heterogeneous manycore architectures based on accelerators integrated into simple cores. We reveal the steps undertaken to integrate the accelerators into an open source core and use them via custom instructions. We automate custom hardware generation to facilitate design space exploration of heterogeneous architectures.

3. Design Approach

Our approach for designing domain-specific heterogeneous architectures was based on instruction augmentation through the integration of hardware accelerators added into simple cores. We based these accelerators on an analysis of the application where hot-spots suitable for hardware acceleration are identified. The overall design flows consisted of the following steps:

- Application development
- Analysis and code generation
- Accelerator integration
- System integration

In the following section, we describe a generic design flow, and then give a description of its realization, starting from CAL dataflow applications in Section 3.2. The generic design flow and its specific realization are illustrated in Figure 1. On the left hand side of the figure, where the generic flow is shown, one can see that the application is fed to the code generation and analysis tools. The analysis tool provides hot-spot information to the code generation tool, which sends feedback data to the developer. The generated hardware code is passed down to an accelerator integration tool to be integrated to a core and form a tile. These tiles are fed to a system integration tool to be connected together with a NoC to form a manycore. The generated software code is passed down to a native compiler. Together with a mapping tool, the compiler maps executables onto the generated architecture. The configuration parameters are used in different tools to determine the features of the hardware components. The right hand side of the figure shows the tools, languages and hardware components used in the realization of the design flow. Note that the system integration step is grayed out and the tools are not yet specified, because this step of the design flow is an ongoing work and is not covered in this study.
Figure 1. Illustration of the design flow and its realization. The grayed out components are not addressed by this study.

### 3.1. Generic Design Flow

The generic design flow consists of the following steps: (1) application development in a suitable programming language; (2) analysis and code generation to generate hardware and low-level software for the intended architecture; (3) accelerator integration where the hardware accelerator is integrated with the basic core; and (4) system integration where all the accelerated cores are integrated and connected through a NoC. This generic description is independent of the programming language, programming model, tool and hardware.

#### 3.1.1. Application Development

The design method aims to build an efficient architecture for a specific domain of applications by integrating task-specific custom hardware into simple cores. The architecture configurations are based on the requirements of the applications. Therefore, the design method starts from the application description.
From an application programming perspective, programming languages in which concurrency is inherent in the language are attracting increased attention in mainstream parallel computing compared to sequential languages. These concurrent languages make explicit the inherent parallelism of the applications. In the proposed method, to target a manycore architecture, the application development should be performed in a programming language that can express different levels of parallelism, such as instruction-level, task-level, and data-level parallelism. This facilitates generation of the task-specific cores, which are the fundamental components of the target architectures.

A developer can use the feedback from the analysis and code generation tools to improve the application. Additionally, the feedback can be used to adjust the application to make it more suitable for use in architecture development, especially in cases where acceleration of the application is desired, but the implementation lacks a distinguishable hot-spot.

3.1.2. Analysis

The most compute-intensive parts of the application (hot-spots) were implemented as custom hardware to be integrated into the base core. To identify the compute-intensive parts of the application, analysis of the application is required (often referred as profiling). Analysis methods are usually divided into static and dynamic analysis methods [55–57]. Static analysis can be used to highlight possible coding errors, and mathematically prove properties of a program via formal methods, or to count the number of operations and operands. It can also be used to estimate execution times for applications which have constant or static behavior. However, dynamic analysis is required to obtain execution measures for applications with dynamic behavior, and to define the most frequently executed, computationally-intensive code blocks. Analysis information useful for hot-spot identification includes the execution rate, the number of operations and operands, the complexity of the operations, and the execution time.

During the analysis, architectural features can also be generated, including memory requirements, communication characteristics, and the required number of cores. These features can be adjusted during code generation for optimization purposes.

3.1.3. Code Generation

Once identified, hot-spots are candidates for implementation as accelerators. These were implemented as custom hardware, which can be performed using a high-level language such as C before using HLS tools to generate the RTL design, or directly, using a hardware description language. The code generation step consisted of both software and hardware generation. Software code generation resulted in native code with embedded instructions for the accelerator. This native code can then be translated to the target machine code using proprietary tools. Hardware generation involved generating hardware descriptions for the custom hardware blocks.

3.1.4. Accelerator Integration

The custom hardware block, which was generated in the previous step, needs to be integrated to the base core so that the hot-spot can be delegated to the accelerator while the rest of the application is executed on the core. The accelerator can be connected to the data bus and be memory mapped, it can be connected through custom interfaces, or even act as an instruction extension. However, instruction extensions need changes in the compiler unless the instructions are already in the instruction set and supported by the compiler. During integration, the custom hardware can be interfaced to memory to provide direct memory access. Moreover, the custom hardware can be connected to a network-on-chip through a network interface, enabling the other cores on the network to make use of the custom hardware. In-short, the integration method can vary based on different aspects, such as the features of the base core, architectural requirements, and application requirements.
3.1.5. System Integration

The prior steps of the design flow produce tiles for individual tasks in the applications. To execute the entire application, or a set of applications, the tiles need to be connected to each other to form a manycore architecture. In particular, for dataflow applications that have significant amount of core-to-core (or tile-to-tile) communication, the tiles must be connected with a proper infrastructure that supports tile-to-tile communication. The architectural information gathered during the analysis and code generation stages was used in this step while configuring the network-on-chip (NoC), and when determining whether to add tiles containing only memory to provide more on-chip memory.

In this step, an important decision is the choice of NoC topology. The efficiency of the topology might change based on the domain of the applications. However, for dataflow applications, we suggest a 2D mesh structure based on our experience during previous work [22,58] with the Epiphany architecture [59]. This structure provides efficient core-to-core communication for dataflow applications in terms of bandwidth and latency. With a proper routing algorithm, such as X-Y routing, it becomes deadlock free. In addition, it is scalable, which enables different sized manycore architectures to be built. However, there are several other NoC topologies, such as torus, ring, star, tree, crossbar, and hypercube [60–62]. A torus has wrap-around connections between the opposite edges of the mesh. However, it would be smart to save the edge connections for extensions to the NoC and external connections. The crossbar topology establishes all-to-all connections, however, this becomes quite expensive in terms of area usage when the size of the network increases. In a star topology, all messages pass through a central router. This router is a potential congestion point, especially for applications with intensive communication needs. In tree topologies, the root node and the nodes close to it become a bottleneck. In the hypercube topology, the number of neighbours of each node is equal to the degree of the topology. To increase the number of nodes in such topologies, one needs to increase the degree of the topology, and hence increase the number of connections to each node. From a scalability point of view, this is an obstacle, as the structure of the NoC needs to be changed with each dimension increase.

3.2. Realization of the Design Flow

We now describe the realization of the design flow in terms of the tools, languages, and other components that have been applied in our proposed approach.

3.2.1. Application Development

Applications that process a continuous stream of data and have high performance requirements, such as video coding/decoding, radar signal processing, and wireless communication, can be implemented efficiently using a dataflow programming model [63]. In this model, applications consist of a number of tasks, which are connected to one another to form the execution flow. They can execute concurrently and express parallelism. We adopted such a dataflow programming model, and developed our applications as streaming applications. An example streaming application can be seen in Figure 2, in which the raw input data flow through different tasks and finally produce the output data. We chose CAL actor language [26] as the application development language because of its concurrent nature, compliance to streaming applications, and ease of use.

CAL [26] is a high-level dataflow programming language [64] that has been adopted by the MPEG Reconfigurable Video Coding (RVC) [65] working group as a part of their standardization efforts. A CAL program consists of actors, which are stateful operators. Actors consist of code blocks called actions that can transform input data streams into output data streams. The state of the actor usually changes while performing the transformation. When there is more than one actor in a program, these actors get connected with channels/ports to form a network. Each actor consumes the tokens on its input ports and produces new tokens on its output ports.
The execution order of the actions can be defined with an explicit schedule in the form of a state machine together with priority definitions for the actions. Additionally, CAL provides a network language to instantiate actors, define channels, and connect the actors via these channels.

Another advantage of using CAL is the opportunity to integrate application analysis and code generation (both software and hardware) in our Cal2Many framework.

![Diagram](image-url)

**Figure 2.** A streaming application example (MPEG-4 Simple Profile Decoder [66]).

### 3.2.2. Analysis

The analysis method and the tools employed depend on the programming language used for application development. For instance, for applications written in C, one can use Valgrind [67] and gnu profiler (gprof) [68] to obtain the total number of instruction fetches per function, the number of function calls, the total execution time of the application and the amount of time spent in each function. We used these tools for one of our Autofocus criterion calculation implementations, which is hand-written in C.

For the applications developed in CAL, we used TURNUS [69], which is a framework for profiling dataflow applications running on heterogeneous parallel systems. This framework implements a CAL dataflow profiler on top of the Open RVC-CAL Compiler (Orcc) [70]. TURNUS provides high level application and architecture modeling as well as tools for application profiling, system level performance estimation, and optimization.

TURNUS provides both static and dynamic analysis information relevant for both the software and hardware aspects, however our focus is on the software aspects. Some of the key analysis information provided by this tool are the firing rates of the actions, the number of executed operations, the input/output tokens consumed and produced, and communication buffer utilization. To obtain dynamic analysis data such as firing rates of the actions, the application was executed at least once. For each executed action, both the computational load and the data-transfers and storage load were evaluated. The computational load, given as weights in Figure 3, was measured in terms of executed operators and control statements (i.e., comparison, logical, arithmetic and data movement instructions). This information was used for hot-spot identification, while the communication information was used for configuring the network-on-chip.

This tool can provide performance estimation based on an architecture model that is an undirected graph where each node represents an operator (a processing element such as a CPU or an FPGA in the terms of [71]) or a communication element (e.g., bus and memories). However, the performance estimation is not critically necessary to identify the hot-spots of the applications. Moreover, we do not want to limit our hot-spot identification to any platform. Therefore, we do not focus on performance estimation.

There are other efforts to profile the applications at different levels such as basic block level in static single assignment (SSA) form to find out the most frequently executed instruction sequences [72,73]. However, we maintained the analysis at a higher level of abstraction to keep it simple and
architecture agnostic. CAL actions are meant to be small code blocks and yet they can be significantly compute-intensive by including heavy floating-point operations such as multiplication, division, and taking a square root. Therefore, analyzing the application at the action level is sufficient to identify hot-spots.

The case studies implemented in CAL were analyzed statically and dynamically via TURNUS. Firing rate of the actions together with the number of executed operations are sufficient to identify the hot-spots. The tool provides weights, which can be interpreted as a compute density, for each action based on their firing rate, number of operations and the number of operands used in the operations. This information can be used directly for hot-spot identification. Figure 3 shows an overview of the actions in the QRD case study and their weights in an actor. One can see that calculate_boundary action is the hot-spot of the given actor despite not being the most frequently executed action. This is due to the number of operations in the actions.

![Figure 3. An overview of firing rates and weights of actions in an actor performing QR decomposition.](image)

Some actors could have dynamic execution paths (different execution order and frequency for the actions) depending on the input tokens or the state of the actor. In such cases, the application can be executed a number of times to obtain an average value for the analysis data.

Gathering the analysis data and identifying the hot-spots are performed automatically by tools. Once the hot-spot actions are identified, they are tagged manually for the code generation.

3.2.3. Code Generation

To automate the code generation, we used our in-house developed Cal2Many framework, which is a source-to-source compiler [20]. Figure 4 gives an overview of the framework. It takes CAL application as input and transforms it into two consecutive intermediate representations, namely Actor Machine (AM) [74] and Action Execution Intermediate Representation (AEIR) [20]. After the transformations and utilization of necessary libraries, it finally generates native code in various target specific languages.

In CAL, actors execute by firing actions, which satisfy the conditions such as the state of the actor, the availability of input tokens, and their values. While generating the code, the framework has to take ordering of these conditions into account together with the number of the tests performed before firing an action. We adapted a simple actor model called actor machine to schedule the testing of the firing condition of an action. To get closer to the target imperative languages, we used the second intermediate representation (AEIR), which is a data structure that allows generating code in imperative languages such as C, C++ and java. It has constructs for function declarations, expressions, statements, and functions calls. Translation of AM to AEIR consisted of two main passes. The first pass dealt with the translation of CAL constructs to imperative constructs including functions, statements, expressions, actions, and variable declaration. The pass took into account the translation of the AM to a sequential action scheduler.

We previously developed four different back-ends, i.e., sequential C for uni-processors, aJava and aStruct for Ambric architecture, native-C for Epiphany architecture, and target specific language
(Scala subset) for ePuma [75] and EIT [76] SIMD architectures, as shown in Figure 4. The C and Chisel hybrid back-end was developed as part of this work.

![Diagram](image_url)  
**Figure 4.** An overview of Cal2Many framework including the back-end for generating custom hardware blocks.

We used the C back-end to generate the software code to be executed on the base RISC-V processor core. We modified the parallel C back-end to generate RISC-V compilable C code, including embedded custom instructions. However, code generation in our design flow consists of two passes. The first pass generates the architecture-specific software code, which is C code with embedded custom instructions provided by the RISC-V ISA. The second part is generating the hardware blocks in Chisel, corresponding to the hot-spots in the CAL application. To generate the hardware blocks, we developed a new back-end that generates a combination of hardware and software code, using the Chisel and C languages, respectively. Those actions, which are tagged as hot-spots during the analysis step, were converted into hardware blocks acting as accelerators in Chisel, whereas the rest of the application was converted into target-specific software, which at present is in C. The code generator makes use of a library of hardware blocks consisting of efficient implementations for floating-point division [77] and square root operations.

The bodies of the tagged actions are replaced with custom instruction calls to trigger the hardware accelerators. The necessary input data are forwarded to the accelerators and the results are read through these instructions. The bit fields of the instruction can be seen in Figure 5, where xs1 and xs2 indicate the usage of source registers, and xd indicates if the accelerator will write a result in the destination register. If the xd bit is set, the core will wait until the accelerator finishes the computation. The generated hardware is later integrated to a rocket core through the rocket chip generator.
Cal2Chisel Back-end

The common language used in the RISC-V environment is Chisel [21], which is an open-source hardware description language developed at UC Berkeley. It is embedded in the Scala language [78]. Some features of the Chisel language are parameterized generators, algebraic construction, abstract data types, interfaces, bulk connections, hierarchical, object oriented and functional constructions and multiple clock domains. The Chisel compiler, which is provided by UC Berkeley, can generate low level synthesizable Verilog code.

The new back-end makes use of the intermediate representations (AM and AEIR) provided by the Cal2Many framework to generate a combination of C (software) and Chisel (hardware) code. The C code for the untagged (non-hot-spot) actions is generated in the pass where the AEIR structure is visited once. However, when the tagged (hot-spot) actions are visited in the AEIR structure, the Chisel code and custom instruction calls in C are generated in two phases. In the first phase, the entire action structure is scanned to generate a static single assignment (SSA) form by generating new versions of variables for each assignment and placing phi functions in necessary places. Wires are defined for the first set of variables and the corresponding Chisel code is generated. Then, information about the usage of actor-wide global variables and input and output ports are gathered for identifying the inputs and the outputs of the accelerator. An instance of the Chisel code for inputs and outputs can be seen as follows:

```scala
val io = IO(new Bundle{
  val r_in = Input(UInt(width = 32.W))
  val x_in_in = Input(UInt(width = 32.W))
  val r_out = Output(UInt(width = 32.W))
  val c_out = Output(UInt(width = 32.W))
  val s_out = Output(UInt(width = 32.W))
})
```

In the second phase, the custom instructions in C/assembly and the rest of the Chisel code is generated. The custom instructions are inlined as assembly code via macros defined in a header file. The inputs and outputs are identified, i.e., the global variables to be read are identified as inputs and the ones to be modified are identified as both inputs and outputs. All variables are defined (except the first set that are already defined in the first phase). While going through the statements, the phi functions are replaced by multiplexers and the common operations such as the floating-point operations and complex number operations are replaced with hardware blocks from a manually-developed library. Finally, the outputs are connected to the final versions of the corresponding variables. In the current implementation of the back-end, loops and arrays are not supported. If only one element of a global array is used in the tagged action, the index will be used out of the hardware block to access the array and the value will be an input to the hardware.

We have experienced that the critical path of the generated hardware in many cases becomes quite long and causes the maximum clock frequency to be significantly lowered. To alleviate this, it is necessary to introduce pipelining in the accelerators. Therefore, a pipelining feature was added to the back-end by using pipelined hardware blocks for arithmetic operations. Delay registers were inserted into the data paths where they need to be synchronized.

![Figure 5. Custom instruction provided by the RISC-V ISA to call the custom hardware blocks.](image-url)
Figure 6 shows the generated C and Chisel code (without delay registers) for a tagged CAL action. In the CAL code, indices are used for accessing the array elements. However, to avoid transfer of the whole array to the accelerator, these indices are moved to the C code and only single array elements are transferred. The custom instruction is called twice with different funct fields (refer to Figure 5) to provide all the inputs to the accelerator. The funct field is used to let the interface know when all the inputs are provided so that they can be fed to the accelerator. The floating-point operations are performed by instantiating the pre-defined hardware blocks such as FPAdd, FPMult and fpSqrt, as seen in the generated Chisel code (Figure 6). The code also shows the connection between the modules and different variables.

3.2.4. Accelerator Integration

We have used the RISC-V environment for our integration step. RISC-V is an open instruction set architecture based on reduced instruction set computing (RISC) principles. It originated at University of California (UC), Berkeley. Several processing cores have been developed that implement this instruction set, such as the Berkeley Out-of-Order Machine (BOOM) [79], rocket core [27], Sodor CPU [80], picoRV32 [81] and scr1 [82]. We chose the rocket core [27] for our case studies as it has an interface to connect custom hardware blocks that enables us to create cores with application-specific augmentations. Additionally, the rocket chip generator produces emulator and synthesizable Verilog code for this core and its surrounding components.

Rocket core [27] is an in-order scalar processor based on the RISC-V ISA. It features a five-stage pipeline and has an integer ALU, an optional FPU and L1 data and instruction caches. This core supports up to four accelerators via an interface called rocket custom co-processor (RoCC) [83], as shown in Figure 7. We can see a tile consisting of the rocket core with L1 cache and an accelerator connected to the core and the cache via the RoCC interface. Custom instructions of the RISC-V ISA can be forwarded to the accelerator through this interface. Depending on a bit field in the custom instruction, the core might halt until it receives a response from the accelerator.
A rocket core can be generated via the rocket chip generator using a Scala program that invokes the Chisel compiler to produce RTL describing a complete system-on-chip (SoC). This generator allows the developers to configure the generated core by changing parameters such as cache sizes, FPU usage, number of cores, and accelerator usage. The Rocket chip environment also includes cycle-accurate Verilator [84] for simulations and a cycle-accurate C++ emulator.

![RoCC Interface](image)

**Figure 7.** A simplified view of the RoCC interface.

The accelerator was integrated into the rocket core using the RoCC interface in several steps. First, the Chisel code for the accelerator was copied into the Chisel source folder of the rocket chip generator. Then, a Chisel class (which becomes a module in Verilog) was added to the rocket core source. This extends the RoCC interface, instantiates the accelerator, and connects the accelerator I/O wires to the RoCC interface I/O wires. This class needs to be instantiated in a configuration class within the configuration file of the rocket core, where the core components can be added, removed or modified. Custom instructions are bound to the accelerators in this configuration class. Each custom instruction can be bound to a different accelerator. Because four custom instructions are supported by RISC-V ISA, the number of accelerators that can be connected to the same core is limited to four. However, this can be increased using the funct field of the custom instruction as an identifier to determine the accelerator to use, while connecting the RoCC interface I/O wires to the accelerator I/O wires. The RISC-V ISA supports two 64-bit source registers in the custom instructions. If the accelerator requires more input data at once, the extended RoCC interface is used for storing the input data until the last inputs arrive and feed all the data to the accelerator at once. The accelerator returns the result through the same interface.

The rocket chip configuration file includes many different configurations to determine the components and their settings to be used in the generated architectures. A new configuration class is needed to instantiate the new rocket core integrated with the accelerator. Once the new configurations are added, the generator can produce a cycle-accurate emulator or synthesizable Verilog code for the new tile consisting of the base core, memory, and the accelerator.

This step of the design flow produces tiles containing different components including processor, accelerator, memory and the connections between these components. Theoretically, the types of the tiles can be:

- Processor Tile, consisting of a processing core, local memory and optionally an accelerator
- Memory Tile, consisting of only memory
- Accelerator Tile, consisting of only an accelerator

The tile types are illustrated in Figure 8. Thus far, the rocket chip generator allows us to generate processor tiles with and without the accelerators. However, we plan to extend the tile generation to cover all types of tiles.
3.2.5. System Integration

The last step of the design flow is system-level integration of the tiles. This step was not realized in this study, hence we do not have a generated manycore architecture. The rocket chip generator has configurations to generate architectures with more than one core. However, it uses cache to connect the cores together. Because this is not as efficient as having a network-on-chip (to handle cases with intense core-to-core communication), we do not generate manycore architectures via the rocket chip generator. This step of the design method is being pursued in our ongoing work with the aim to add support for a network-on-chip to the rocket chip generator.

4. Case Studies and Implementations

This section provides the details of the implementation of the case studies Autofocus criterion calculation and QR decomposition.

The case studies are considered as a proof-of-concept for the design method, and to evaluate the hardware back-end of the Cal2Many framework. The first case study is QR decomposition, which is implemented in CAL actor language. The second case study is an Autofocus criterion calculation using cubic interpolation. Different versions of this application are implemented in sequential and dataflow fashions, in the C and CAL languages, respectively. The C version is implemented as a proof-of-concept, whereas the CAL version is implemented to evaluate the automatic code (C and Chisel) generation with a more complicated accelerator. The Cal2Many framework is used for automatically generating C and Chisel code for the CAL implementations of the case studies. Single tiles are produced at the end of each case study.

In the case studies, where the accelerators are automatically generated, hand-written accelerators are also developed, and used as reference implementations for evaluation. The accelerators are integrated into a rocket core through the RoCC interface. Verilog code for the accelerators is generated and synthesized using Xilinx tools. Additionally, rocket cores with and without the accelerators are synthesized individually. The accelerators are integrated to the rocket core in three steps:

- Connect the RoCC interface to the I/O of the accelerator.
- Adjust the core configuration and binding the accelerator to a custom instruction.
- Adjust the platform configuration to use the new core configuration which includes the accelerator.

The following sections provide details of the case studies and their corresponding implementations.
4.1. QR Decomposition

QR decomposition (QRD) or QR factorization is decomposition of a matrix into an orthogonal matrix $Q$ and an upper triangular matrix $R$ \[85\]. The decomposition equation for a square matrix $A$ is simply $A = QR$. The matrix $A$ does not necessarily need to be square. The equation for an $m \times n$ matrix, where $m \geq n$, is as follows:

$$A = QR = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1.$$

QRD is used in numerous applications for replacing matrix inversions to avoid precision loss and reduce the number of operations. It is also a part of the solution to the linear least squares problem and the basis of an eigenvalue algorithm (the QR algorithm). There are several different methods to perform QRD, such as the Givens Rotations, Householder and Gram–Schmidt methods \[22\].

The method employed in this work for decomposition is Givens Rotations with a systolic array implementation \[86\]. The structure of the parallel implementation is given in Figure 9. There are 10 actors, consisting of two different types, namely boundary and inner actors. These actors can be mapped onto separate cores. However, in this case study, they are manually converted into actions to be combined in a single actor and executed on a single core. When these actors are combined, a few further control actions are required to help with communication and scheduling. All of the actions can be seen in Figure 3.

The following computations are performed in `calculate_boundary` and `calculate_inner` actions.

```plaintext
calculate_inner:
\[
\begin{align*}
    r &= s \times x + c \times r \\
    x &= c \times x - s \times r
\end{align*}
\]
```

```plaintext
calculate_boundary:
\[
\begin{align*}
    c &= r / \sqrt{r^2 + x^2} \\
    s &= x / \sqrt{r^2 + x^2} \\
    r &= \sqrt{r^2 + x^2}
\end{align*}
\]
```

![Figure 9. Systolic Array implementation of Givens Rotations (QRD). The arrows show the direction of the data movement.](image)

According to the analysis tool, the `calculate_boundary` action is the hot-spot of the application, as shown in Figure 3. Therefore, Chisel code is generated for this action while C code is generated...
for the remainder of the application. The hardware for the action is also implemented manually as a baseline for comparison.

RISC-V instructions support one destination register. However, the QRD accelerator returns at least three floating-point numbers, thus requiring at least two 64-bit registers. Therefore, the custom instruction is called twice for firing the hand-written accelerator: once for sending the input data to the accelerator and reading the first two results and once for reading the last result. These operations are distinguished via the \textit{funct} bits of the custom instruction within the extended RoCC interface that stores the last result until the second instruction call. The generated C + Chisel accelerator requires three custom instruction calls because it has five floating-point inputs and three outputs. The custom instruction is called twice to send the inputs, and a further time to read the last output.

4.2. Autofocus Criterion Calculation in Synthetic Aperture Radar Systems

Synthetic-Aperture Radar (SAR) systems produce high resolution images of the ground. The back-projection integration technique has been applied in SAR systems, enabling processing of the image in the time domain, which makes it possible to compensate for non-linear flight tracks. However, the cost is typically a high computational burden. In reality, the flight path is not perfectly linear. This can, however, be compensated for in the processing. The compensations are typically based on positioning information from GPS. If this information is insufficient, or even missing, autofocus can be used. The autofocus calculations use the image data itself, and are performed before each subaperture merge. One autofocus method, which assumes a merge base of two, relies on finding the flight path compensation that results in the best possible match between the images of the contributing subapertures in a merge. Several flight path compensations are thus tested before a merge. The image match is checked according to a selected focus criterion. The criterion assumed in this study is maximization of correlation of image data. [24]. As the criterion calculations are carried out many times for each merge, it is important that these are done efficiently. The calculations include interpolations and correlation calculations.

The autofocus implementation takes $6 \times 6$ blocks of image pixels as input and applies complex cubic interpolations to these blocks. The hot-spot of the application is identified as the computation of the cubic interpolation. This computation is a combination of complex multiplication, complex subtraction and complex division. Each of these operations is implemented as a building block. Figure 10 presents the structure of the complex division block, consisting of floating-point operations. This block is used as one of the building blocks while developing the accelerator, and it is represented as \texttt{CDiv} in Figure 11 that shows the fully flattened version of the accelerator.

![Figure 10. Building block that performs division on complex numbers.](image-url)
The mathematical expressions of the operations performed during the cubic interpolation are as follows:

\[ d_i = x_{\text{int}} - x_i \quad i \in \{0..3\} \]

\[ p_{01} = \frac{(y_0 \times d_1) - (y_1 \times d_0)}{x_0 - x_1} \]

\[ p_{12} = \frac{(y_1 \times d_2) - (y_2 \times d_1)}{x_1 - x_2} \]

\[ p_{23} = \frac{(y_2 \times d_3) - (y_3 \times d_2)}{x_2 - x_3} \]

\[ p_{02} = \frac{(p_{01} \times d_2) - (p_{12} \times d_0)}{x_0 - x_2} \]

\[ p_{13} = \frac{(p_{12} \times d_3) - (p_{23} \times d_1)}{x_1 - x_3} \]

\[ p_{03} = \frac{(p_{02} \times d_3) - (p_{13} \times d_0)}{x_0 - x_3} \]

where \( x_{\text{int}} \) is a constant, \( x \) and \( y \) are the positions and values of the input pixels, and \( p_{03} \) is the computed result. The Red highlighted box in Figure 11 performs the operations to calculate \( p_{01} \) and the small red boxes represent delay registers.

Figure 11. Structure of the flat cubic interpolation accelerator.

We implemented the autofocus criterion calculation in CAL actor language and generated the software and hardware through our tool-chain. Furthermore, we manually implemented the software and the hardware of the application in native C and Chisel languages, respectively. These implementations will be referred to as Generated and Hand-written in the rest of the paper.

The Generated and Hand-written implementations have different designs. Figure 12 presents the dataflow diagram of the CAL (Generated) implementation. One can see that the main function (cubic) is implemented as an actor and instantiated several times to exploit data-level and task-level parallelism. However, since our tool-chain supports composition of actors [87] and we are intending to generate code for single-core execution, therefore, all the actors of the autofocus criterion calculation (seen in Figure 12) are composed into a single actor to be mapped on a single core. This operation results in a more complex scheduler in the composition, and requires handling actor-to-actor communication. To analyze the CAL actor, the TURNUS framework is used, and cubic interpolation is identified as the hot-spot.
The cubic interpolation of autofocus criterion calculation is implemented in CAL as a action that performs only a single step of the interpolation (calculation of a single $p$ value). This helps to keep the action simple and decreases the number of global variables that provide the input values to the interpolation kernel. The action is executed six times in a row to complete the interpolation of 4 pixels and produce a single pixel result.

The $\text{Generated}$ accelerator makes use of complex number arithmetic blocks implemented manually with pipelines. The number of used complex and floating-point blocks of the accelerators are given in Tables 1 and 2. The generated accelerator results are equal to the results given for the $\text{Folded Accelerator}$ in the tables. The generated accelerator requires 4 complex numbers (each of 64-bits) as inputs to be fired. Each complex number has two floating-point values representing the real part and the imaginary part. The total number of inputs to the accelerator is 8 floating-point numbers, represented in IEEE-754 single precision (binary32) format [88]. The RoCC interface supports transfer of two 64-bit values from core to the accelerator with each custom instruction. Therefore, we combine real and imaginary values of each complex number in a 64-bit register and send 4 floating-point numbers to the accelerator with each instruction. Hence, two custom instruction calls are required to transfer four complex numbers to the accelerator. The accelerator is pipelined and has 20 stages. We have extended the RoCC interface to handle synchronization of the input data between the instruction calls for firing. The extended interface stores the input data until it receives the instruction with the last data required. Then, all of the input data are fed to the accelerator together with an enable signal that fires the accelerator.

Two different versions of the $\text{Hand-written}$ accelerator were developed. The first version of the cubic interpolation accelerator, which is flat and presented in Figure 11, consists of six identical structures—one of them is highlighted in Figure 11. This hardware accelerator is identical to the cubic interpolation function of the hand-written C implementation, and requires 4 pixels as input, with each pixel consisting of two complex numbers (value and position). These inputs are named as $x$ and $y$ in Figure 11. It takes four instructions to send all the input data.

The second version of the hand-written accelerator is optimized (folded) for resource usage. It uses only one of these six identical structures in a software controlled loop and the four individual $\text{CSub}$ blocks to perform the cubic interpolation, as shown on the right hand side of the Figure 11.
Similar to the generated accelerator, this accelerator is fired six times by the software to calculate one result. To be fired, this accelerator requires four complex numbers, which takes two instructions.

The flat accelerator is designed with a fully pipelined structure and can produce one result (a complex number) per cycle. However, due to the limitations in the data transfer, it can be fired at every fourth cycle (ignoring the cache misses). The folded accelerator also has a pipelined structure and can be fired at every cycle. However, input limitations mean that it can only be fired every other cycle. Moreover, with the folded accelerator, the first three iterations of the loop for calculating one final result can be executed two cycles apart (without waiting for the result of previous iterations). However, synchronization problems may occur with the results returned from the accelerator if the core does not halt until the arrival of the result.

Table 1 presents the number of (blocks) complex operations executed by the accelerators, whereas Table 2 shows the number of floating-point operations performed by each block and the accelerators. The generated accelerator and the folded hand-written accelerator have the same results and are presented in Folded Accelerator column. In total, 140 floating-point operations are performed for each set of inputs (4 pixels). Floating-point multiplication and division operations employ integer multiplications. To perform these integer multiplications on-chip, DSP blocks are utilized.

**Table 1. Number of complex operations in the accelerators.**

<table>
<thead>
<tr>
<th></th>
<th>Flat Accelerator</th>
<th>Folded Accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMul</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>CDiv</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>CSub</td>
<td>16</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 2. Number of floating-point operations in blocks and the accelerators.**

<table>
<thead>
<tr>
<th></th>
<th>Flat Accelerator</th>
<th>Folded Accelerator</th>
<th>CMul</th>
<th>CSub</th>
<th>CDiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPMul</td>
<td>84</td>
<td>14</td>
<td>4</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>FPDiv</td>
<td>12</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>FPSub</td>
<td>50</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FPAAdd</td>
<td>24</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

5. Results

This section provides the performance, resource utilization and timing results for the case studies considered in this work. The performance results are obtained through executing the case studies on the cycle-accurate emulators generated with the rocket chip generator. Hardware counters are used to obtain the cycle counts. Resource usage results are provided by Xilinx synthesis tools. The target platform is the Xilinx VCU108 evaluation kit, including the Virtex UltraScale XCVU095 FPGA.

5.1. QR Decomposition

The QRD case study takes a matrix of $16 \times 16$ elements as input data. Table 3 present the performance results of different implementations of the QRD in terms of achieved clock frequency and cycle count. The first implementation results are for software generated by the Cal2Many framework and executed on the rocket core without using any custom hardware. The second row of results correspond to the rocket core and accelerator where both the software for the core and hardware for the accelerator are generated by the Cal2Many tool-chain. Finally, the last set of results are for the generated software code executing on rocket core along with a hand-written custom hardware implementation.
Table 3. Performance results for QRD in terms of cycle count and clock frequency.

<table>
<thead>
<tr>
<th></th>
<th>Cycle Count</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>366 k</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Generated Accelerator</td>
<td>92 k</td>
<td>56 MHz</td>
</tr>
<tr>
<td>Rocket Core + Hand-written Accelerator</td>
<td>89 k</td>
<td>56 Mhz</td>
</tr>
</tbody>
</table>

It is apparent from the results that could achieve similar clock frequencies for all three cases, primarily because of the pipelining incorporated in the Cal2Chisel back-end that results in generating pipelined accelerators. The combination of accelerator and the rocket core outperforms the single-core execution by a factor of 4. However, there is a small drop of 4% in the performance of the generated accelerator with respect to the hand-written accelerator. The main reason for this difference the different the number of inputs. During code generation, use of the global array indices is moved out of the accelerator. However, they are still passed to the accelerator as inputs. The hand-written accelerator takes in two floating-point inputs requiring a single custom instruction call, whereas the generated accelerator takes in five floating-point inputs, requiring at least two custom instruction calls. Copying the input variables to the source registers adds extra cost. This difference can be reduced further by incorporating optimizations in the hardware code generation.

The resource usage and clock frequency results for the rocket core and the accelerators can be seen in Table 4. The main reason for the difference in look-up tables and flip-flops in the two accelerator implementations is the lack of support for a mixture of integer/floating-point operations. The code generation tool treats integer operations similarly to floating-point operations, and utilizes floating point operation blocks. Specifically, there is an integer addition operation that is converted into a floating-point operation in the generated accelerator, which causes an increase in resource usage. Thus, there is potential for further improvement in the resource usage of the generated hardware. The maximum clock frequency of the generated accelerator is 20 MHz without the pipelining feature. However, by inserting the pipeline stages, the maximum clock frequency increases to 104 MHz, equivalent to the clock frequency of the hand-written accelerator. The accelerators use the same hardware library for the arithmetic operations, which plays a significant role in determining the clock frequency.

Table 4. Resource usage and timing results of the rocket core and QRD accelerators

<table>
<thead>
<tr>
<th></th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>39,843</td>
<td>16,512</td>
<td>12</td>
<td>24</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Generated Accelerator</td>
<td>1165</td>
<td>914</td>
<td>2</td>
<td>25</td>
<td>104 MHz</td>
</tr>
<tr>
<td>Hand-written Accelerator</td>
<td>999</td>
<td>812</td>
<td>2</td>
<td>25</td>
<td>104 MHz</td>
</tr>
</tbody>
</table>

In addition, the accelerator is fully pipelined and parallelized, meaning it can produce one set of results per cycle. However, due to limitations in the data movement between the core and the accelerator, it is not possible to make use of this feature. The standalone latency of the accelerator is 14 cycles and the total latency (including data movement) is approximately 27 cycles.

To summarize, we are able to achieve an improvement of $4 \times$ in performance with respect to rocket core software execution, and the overhead of generated accelerator vs hand-written accelerator is minimal.

5.2. Autofocus Criterion Calculation

The the autofocus criterion calculation application, consisting of integrated cubic interpolation accelerators, is tested using 12 kernels, each consisting of $6 \times 6$ image pixels. Cubic interpolation is applied 324 times within the autofocus criterion calculation application.
Table 5 shows the performance results for the autofocus criterion calculation in terms of cycle count and achieved clock frequencies. All of these implementations use hand-optimized software. There are two implementations of the hand-written accelerators, namely: flat and folded. The flat accelerator is fully parallelized and has the best performance in terms of cycle count. On the other hand, the folded hand-written implementation, which applies the same cubic implementation block in a loop, has similar a performance to the generated accelerator. Clearly, all of the implementations with integrated accelerators outperform the software execution on rocket core by a factor of 2.8–4.8.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cycle Count</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>306 k cycles</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Hand-written Flat Accelerator</td>
<td>64 k cycles</td>
<td>56 MHz</td>
</tr>
<tr>
<td>Rocket Core + Hand-written Folded Accelerator</td>
<td>108 k cycles</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Generated Accelerator</td>
<td>108 k cycles</td>
<td>58 MHz</td>
</tr>
</tbody>
</table>

The hand-written flat accelerator implementing cubic interpolation produces a final result in 52 cycles. Due to the pipelined structure, the accelerator can produce one result every clock cycle, provided all the required inputs are loaded. Since the RoCC interface can provide at most four floating-point inputs per-cycle, the accelerator can produce four result per four cycles. Furthermore, the rocket core waits until the accelerator produces its result. Hence, the accelerator can compute one result in 52 cycles. When the accelerator is integrated into the core, it takes eight cycles to copy the data into the source registers and fire the accelerator. Additionally, it takes two cycles to read the result. Therefore, in total, it takes 62 cycles to compute a result. While measuring these results, the caches are pre-filled, hence there are no cache misses.

The hand-written folded accelerator and the generated accelerator, both of which perform cubic interpolation, are similar in their design. They both compute one of the six identical functions shown in Figure 11 and produce an intermediate result in 20 cycles. If the data movement overhead is ignored, this accelerator can calculate the final result in six iterations, taking 120 cycles. However, for each firing of the accelerator, data movement costs six additional cycles (four for filling the source register and two for reading the destination register). Thus, each iteration takes 26 cycles, meaning that the whole autofocus criterion computation takes 156 cycles (ignoring any cache misses).

Table 6 provides synthesis results for individual rocket core, individual accelerator implementations and integrated (core + accelerator) designs. The results for the integrated designs include the resource usage of the extended RoCC interfaces. The hand-written folded and the generated accelerators use only one of the six main blocks, highlighted with dashed lines in Figure 11. Therefore, one would expect a $6\times$ reduction in resource usage compared to the hand-written flat implementation. However, the CSub components, which produce the $d$ values, are used in the folded version as well. Hence, LUT and FF usage do not decrease exactly by six times. However, one can see that the DSP and BRAM usages actually drop to 1/6, as they are only used in the main blocks.
Table 6. Resource usage and timing results for the rocket core and the integrated designs including hand-written and generated accelerators.

<table>
<thead>
<tr>
<th></th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>39,843</td>
<td>16,512</td>
<td>24</td>
<td>12</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Hand-written Flat Accelerator</td>
<td>24,707</td>
<td>14,471</td>
<td>252</td>
<td>3</td>
<td>126 MHz</td>
</tr>
<tr>
<td>Rocket + Hand-written Flat Accelerator</td>
<td>68,860</td>
<td>36,722</td>
<td>276</td>
<td>16</td>
<td>56 MHz</td>
</tr>
<tr>
<td>Hand-written Folded Accelerator</td>
<td>5187</td>
<td>3732</td>
<td>42</td>
<td>0.5</td>
<td>131 MHz</td>
</tr>
<tr>
<td>Rocket + Hand-written Folded Accelerator</td>
<td>46,124</td>
<td>21,518</td>
<td>66</td>
<td>14</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Generated Accelerator</td>
<td>5239</td>
<td>3220</td>
<td>42</td>
<td>0.5</td>
<td>117 MHz</td>
</tr>
<tr>
<td>Rocket + Generated Accelerator</td>
<td>44,767</td>
<td>20,113</td>
<td>58</td>
<td>14</td>
<td>58 MHz</td>
</tr>
</tbody>
</table>

The increase in the size of the integrated design causes the critical path of the processor to increase insignificantly. When integrated, the flat accelerator causes the max clock frequency of the processor to decrease from 58 MHz to 56 MHz, whereas the folded and generated accelerators do not result in any reduction of the clock frequency. When synthesized individually, hand-written flat, hand-written folded, and generated accelerators achieve 126 MHz, 131 MHz and 117 MHz clock frequencies, respectively.

We have also automatically generated software code from the CAL program and executed it separately on the rocket core and on the integrated design (rocket core + accelerator). The software execution on the rocket core, take 607 k cycles. The use of the generated accelerator reduces the execution time to 196 k cycles. The main reason is that the hand-written (C) implementation fires the accelerator six times in a row (in the same function). However, the generated (CAL) implementation fires the accelerator once with each function call and returns to the scheduler, which is a state machine implemented as if-else statements. The variables copied into the source registers are global variables and it takes four instructions to move them to the source registers in the generated implementation, whereas in the hand-written implementation the variables copied into the source registers are sent as arguments/parameters to the function and it takes two instructions to move them to the source registers. The CAL implementation consists of many actors and would benefit from being mapped onto different cores. The automated composition of the actors cause a significant performance reduction, which will be our next goal for improving software generation.

To summarize, the hand-written flat accelerator increases the performance of the rocket core by a factor of 4.8 at the cost of a significantly larger resource usage. However, the hand-written folded and the generated accelerators increase the performance by $2.8 \times$ with a very modest increase in the resource usage.

6. Conclusions

Manycore architectures are emerging to meet the performance demands of today’s complex applications. These architectures usually consist of identical or generic cores. However, applications usually consist of a variety of tasks. To achieve the highest efficiency, there is a need to specialize the cores for individual tasks and introduce heterogeneity. However, designing such an architecture is a challenging task.

In this paper, we propose an approach for designing domain-specific heterogeneous tiles based on instruction augmentation through integrating custom hardware to simple cores. The main purpose of generating these tiles is to use them to build future manycore architectures. The design approach aims to build the architecture based on the requirements of the applications within a domain. We developed a tool to automatically generate the custom hardware blocks from CAL applications. We generated custom hardware that is then integrated into a core that executes the RISC-V instruction set. We evaluated our approach by implementing QR decomposition and an autofocus criterion calculation case studies in CAL actor language. The case studies revealed that RISC-V custom instructions can be used for integrating specialized custom hardware to boost performance. The processing cores with custom
hardware outperform the cores without custom hardware by a factor of 2.8–4.8. Additionally, it is shown that automated hardware generation increases the area of the hardware by approximately 0–12% and the performance drop (in terms of cycle count) is 0–9%, which can be decreased even further by optimizing the code generation tool. The pipelining stages implemented during hardware generation play a significant role in improving the maximum clock frequency for the accelerators. However, when integrated with the rocket core, the frequency is saturated to the maximum clock frequency of the rocket core i.e., 56 or 58 MHz.

The significant advantage of our design approach is in terms of productivity, as the resulting accelerator can be produced from the same dataflow program without committing any extra effort into developing the hardware. Manycore architectures are slowly evolving in the direction of specialized cores, and we believe that the proposed approach will facilitate the design of new architectures based on specialized cores.

In the future, we plan to generate a manycore architecture by integrating many tiles with a network-on-chip. While doing this, we will use several case studies to generate the tiles to support a set of applications. Later, we aim to extend the hardware block library and fully automate every design step, including system integration.

**Acknowledgments:** This research was performed within the HiPEC (High Performance Embedded Computing) and NGES (Towards Next Generation Embedded Systems: Utilizing Parallelism and Reconfigurability) projects, which are funded by the Swedish Foundation for Strategic Research (SSF) and a VINNOVA Strategic Innovation grant, respectively. The funds for the projects cover the costs of publishing this research in open access journals. We would like to thank Struan Gray for his contributions with the language of the paper and to Schuyler Eldridge for his help with the accelerator integration.

**Author Contributions:** Suleyman Savas developed the design method and fine tuned it with feedback from the other authors. The experiments were conceived by all authors. Suleyman Savas designed and performed the experiments. The results were analyzed by all authors. Suleyman Savas led the paper writing with contributions from the other authors.

**Conflicts of Interest:** The authors declare no conflict of interest.

**References**


12. Mittal, S. A survey of techniques for architecting and managing asymmetric multicore processors. ACM Comput. Surv. (CSUR) 2016, 48, 45. [CrossRef]


Appendix E

Paper V

Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit

Süleyman Savas, Zain Ul-Abdin, Tomas Nordström, and Yassin Atwa

accepted to International Symposium on VLSI Design (ISVLSI), Miami, Florida, USA, July 15-17, 2019.
Abstract—This paper proposes a novel method for performing square root operation on floating-point numbers represented in IEEE-754 single-precision (binary32) format. The method is implemented using Harmonized Parabolic Synthesis. It is implemented with and without pipeline stages individually and synthesized for two different Xilinx FPGA boards.

The implementations show better resource usage and latency results when compared to other similar works including Xilinx intellectual property (IP) that uses the CORDIC method. Any method calculating the square root will make approximation errors. Unless these errors are distributed evenly around zero, they can accumulate and give a biased result. An attractive feature of the proposed method is the fact that it distributes the errors evenly around zero, in contrast to CORDIC for instance.

Due to the small size, low latency, high throughput, and good error properties, the presented floating-point square root unit is suitable for high performance embedded systems. It can be integrated into a processor's floating point unit or be used as a stand-alone accelerator.

I. INTRODUCTION

Floating-point square root is a significant, fundamental arithmetic operation used by a vast number of applications including radar and baseband signal processing. For instance, QR decomposition [1]–[4], which is one of the major factorizations in linear algebra and used in Massive MIMO [5], performs significant amount of square root operations on floating-point numbers. However, it is quite challenging to implement efficient floating-point arithmetic in hardware. Square root itself is one of the most challenging basic operations to implement among the others such as addition, multiplication, and subtraction. It requires larger area and usually achieves relatively lower performance. Additionally, aforementioned operations are computed as approximations, which introduces errors in the results. These errors may accumulate and cause accuracy degradation in the final result unless the errors are evenly distributed around zero.

In this paper, we implement a single-precision square root hardware that performs square root on floating-point numbers represented in IEEE-754 standard format [6]. The implementations are realized and tested on two field-programmable gate arrays (FPGA) for validation and evaluation.

The proposed method relies on approximation that adopts Harmonized Parabolic Synthesis [7], [8]. It converges faster and provides faster computation and smaller chip area when compared to the other methods such as CORDIC [9], [10] and Newton Raphson [11], [12]. Additionally, in contrast to the other methods, the distribution of the error can be tailored to get an even distribution around zero. Such a distribution improves the accuracy and performance of the remaining of the algorithm that includes the square root. In the Harmonized Parabolic Synthesis methodology the accuracy is fine tuned by using intervals. The accuracy increases with the number of intervals and for single-precision floating-point square root, 64 intervals are sufficient. This method can be used for computing unary operations such as trigonometric functions, logarithms, and square root as well as binary functions such as division [7], [8], [13].

We have used the square root hardware, that is implemented in this paper, in an accelerator utilized for accelerating QR decomposition. The accelerator uses single-precision floating-point numbers and it is integrated to a RISC-V [14] core to build a heterogeneous tile [15]. It performs the square root operation on single-precision floating-point numbers represented in IEEE-754 standard format.

The rest of the paper is structured as follows: Section II provides the necessary background knowledge to understand the rest of the paper. Section III presents other related works on square root operation. Section IV goes through the methodology whereas section V gives the details of the implementation on FPGAs. Results of the implementations are presented in section VI with a discussion. Section VII finalizes the paper with concluding remarks.

II. BACKGROUND

In this section, we introduce the IEEE standard for floating-point number representation and give a brief overview of the square root methods.

A. Binary32 - Single-precision floating-point numbers

There are many floating-point number representations, however, the IEEE Standard for Floating-Point Arithmetic (IEEE-754) [6] is the most common one. This standard defines the representation of floating-point numbers and the arithmetic operations. In this paper, we focus on binary32 numbers, which are more commonly known as single-precision floating-point numbers.
The IEEE 754 standard specifies a binary32 (single-precision floating-point) number as having:

- Sign bit \( s \): 1 bit
- Exponent width \( e \): 8 bits
- Significand (mantissa) \( m \): 24 bits (23 explicitly stored)

as illustrated in Figure 1.

![Sign Exponent Significand (Mantissa)](image)

Fig. 1. Single-Precision (binary32) floating-point representation in IEEE-754 format.

The sign bit determines the sign of the number. The exponent is an 8-bit unsigned integer and biased by +127. The true mantissa consists of 23 visible bits to the right of the decimal-point and 1 invisible leading bit to the left of decimal-point which is 1 unless the exponent is 0. The real value is calculated with the following formula:

\[
(-1)^s \times (1 + m) \times 2^{e-127}
\]

The exponent ranges from \(-126\) to 127 because \(-127\) (all zeros) and 128 (all ones) are reserved and indicate special cases. This format has the range of \(\pm 3.4 \cdot 10^{38}\).

### B. Floating-point square root

There are several algorithms for calculating the square root of a number dating back to Babylonians. Some of these algorithms are the Babylonian method [16], Bakhshali method [17], Newton-Raphson method [18] and, Taylor series expansion [19]. These methods are iterative methods and generally yield approximate results, but can be made arbitrarily precise by increasing the number of calculation steps. Usually, the square root algorithms require an initial seed value. If the initial seed value is far away from the actual square root, the algorithm will be slowed down. It is therefore useful to have a rough estimate, which may be inaccurate but easy to calculate. Our method uses a set of estimates in look-up tables and utilizes auxiliary functions to increase the accuracy of the estimations.

### III. RELATED WORKS

There are several other studies aimed at implementing square root operation efficiently targeting FPGAs. We will mention a few of these studies, which use single-precision floating-point units. The related FPGA implementations usually use older FPGA platforms. Hence it is difficult to make a very accurate comparison between our implementation and the others. However, we still compare resource usage and clock frequency numbers to provide an estimation to the reader.

Suresh et al. [20] and Patil et al. [21] use a non-restoring iterative approach to implement floating-point square root operation on FPGAs and ASICs. In iterative approaches, the amount of resources and clock cycles (latency) increase rapidly with the word size of the input. However, our approach requires fewer resources and cycles to produce a result. For instance, to produce a single result, the mentioned implementations require 40 and 31 cycles, while achieving 205 and 240 MHz clock frequencies respectively as seen in Table II.

Dinechin et al. [22] use a multiplicative (polynomial approximation) method to implement square root operation with different precisions on FPGAs. Since our method is multiplicative as well, their results are close to our results, however, they claim that the cost of correct round is rather high. Hence our resource usage results are lower than what they achieve for correctly rounded, single-precision floating-point results.

Xilinx uses CORDIC method for their square root IP [23]. The achieved clock frequency is higher than what we achieve due to not using DSP and BRAM blocks and consequently having shorter pipeline stages. However, our implementation outperforms the IP in resource usage and latency.

Hasnat et al. [24] develop an FPGA implementation that calculates single-precision floating-point square root and inverse square root simultaneously with Quake’s algorithm [25] modified using Newton Raphson method. They achieve quite low numbers for resource usage on Virtex 5, however, they do not support pipelining and do not provide the number for DSP usage. They use 7 multipliers and each seems to have 23 bits inputs. Thus we assume 2 DSPs are used for each multiplier.

The novelty of this work is the usage of the Harmonized Parabolic Synthesis method for performing square root on single-precision floating-point numbers. The hardware implementation uses three look-up tables, 5 integer multipliers and 7 integer adders, all with different word lengths. The look-up tables consist of 64 words with 28, 22 and 16 bits of word length respectively. These are significantly smaller when compared to other table look-up methods for the same accuracy. Additionally, the other works neglect the error distribution aspect, which can cause trouble due to accumulated errors when the calculation is part of a larger algorithm. Our methodology is able to provide a normal distribution of the error around zero to avoid possible accumulated error problem.

### IV. METHODOLOGY

In this section we will go through a brief description of the mathematics behind the proposed method [8], how to apply it to IEEE-754 single-precision floating-point numbers and the accuracy and error behavior of the method.

#### A. Approximation

The proposed method performs a multiplicative approximation of the square root function and it can be classified as table look-up with auxiliary functions. In order to facilitate the calculations, the range of input and output numbers are constrained to \([0,1]\).

The approximation is calculated as the product of two sub-functions; \(s_1\) and \(s_2\). The first sub-function is a second order polynomial, whereas the second sub-function is a second order interpolation. The first sub-function roughly approximates the original function (square root) while the second sub-function...
improves the approximation. The first sub-function $s_1(x)$ is defined as:

$$s_1(x) = x + c_1(x - x^2),$$

(2)

where $c_1$ is a coefficient. The second sub-function $s_2(x)$ is a second degree interpolation of a help function $f_{\text{help}}(x)$, defined as:

$$f_{\text{help}}(x) = f_{\text{org}}(x)/s_1(x),$$

(3)

where $f_{\text{org}}$ is the original function. Splitting the help function into $N$ intervals provides a set of equations shown below:

$$s_{2,i}(x) = l_{2,i} + j_{2,i}x_w - c_{2,i}x^2_w,$$

(4)

where $0 \leq i < N$, $w = \log(N)$ and $x_w$ is fractional part of $2^w x$.

**B. Application of the Approximation**

In order to facilitate the usage of this method on floating-point numbers, two more steps are added. Thus, the final method consists of three steps namely pre-processing, processing and post-processing as seen in Fig. 2.

![Fig. 2. The steps of the square root method.](image)

In the pre-processing step the exponent is divided by two get the square root as follows:

$$\sqrt{2^e \times M} = 2^{e/2} \times \sqrt{M}$$

(5)

where $e$ is the unbiased exponent and $M$ is the mantissa with the hidden bit. If the exponent is odd then it is reduced by 1 to get an integer after the division. In this case, the mantissa is multiplied by 2 to compensate the reduction in the exponent. Initially the input mantissa (including the hidden bit) has a value between 1 and 2. After the adjustments on the exponent and the mantissa the new range becomes [1, 4]. In order to move the range of the mantissa to [0, 1), which is required by the approximation method, the mantissa is reduced by 1 and then divided by 3 as seen in Figure 6 (we perform some bit tweaking in the circuit to reduce the resource usage). These operations correspond to converting the number to base 4 [26].

In the processing step, the approximation methods (the parabolic synthesis and second degree interpolation) are applied to the mantissa of the input to approximate the square root. The result of this step is the product of the sub-functions.

In the post-processing step, the exponent is transformed back into IEEE 754 format by adding 127 to its value and the final result is produced. A detailed diagram of the method is given in Figure 3. The re-normalization corresponds to pre-processing, whereas approximation is the processing step and the operation on the exponent is the post-processing step.

**C. Accuracy and Error Behavior**

In total, the required number of operations are 5 integer multiplications, 7 integer additions and 1 shift operation. The number of operations do not change with the size of the input value. The same method can be used for arbitrary number of accuracy bits. The size of the operators and look-up tables would increase in case of wider inputs and higher accuracy. Intermediate and final results of the operations are truncated, which means they are rounded to zero to get the smallest error when compared to other rounding methods defined by the IEEE binary32 standards such as 'round to $+\infty$' or 'round to nearest' [6]. Hence 'round to zero' and naturally 'round to -$\infty$' are the only supported rounding methods.

The square root result is tested for $2^{23}(8,388,608)$ inputs (every possible mantissa value) twice; once for odd exponent case and once for even exponent case. The rounding errors for the tested inputs are smaller than $1 ULP$ (unit in the last place) [27], that is used as a measure of accuracy in numeric calculations and calculated as follows:

$$ULP(x) = \epsilon \times 2^{c'},$$

(6)

where $x$ is the result in single-precision floating-point format, $\epsilon$ is machine epsilon and $c'$ is the unbiased exponent of $x$.

When performing computations, the accumulated error in the computations can corrupt the result rapidly. When comparing different approximation method, it should be noted that, to achieve the same performance with the same algorithm, different methods may require different precisions. Approximation methods should therefore be compared in the same context. As an illustration, a comparison is performed between typical distributions of the error of the CORDIC methodology and the Harmonized Parabolic Synthesis method. In Figure 4 the distributions of the error of an implementation using the CORDIC method is shown.

As can be seen in Figure 4 the error is not evenly distributed around zero, which will have devastating effect on
fig. 4. a typical distributions of the error of the CORDIC method.

The problems with accumulated error. The structure of the CORDIC algorithm prevents any significant improvement to the distribution of the error. As shown in [28], CORDIC requires several extra bits of accuracy in order to cope with the error in the approximation.

A significant advantage with the Harmonized Parabolic Synthesis methodology is that the distribution of the error of the approximation can be fully controlled [8]. To minimize the scattering effects on the distribution of the error it is important that it is evenly distributed around zero and with an advantageous shape of the distribution. Because of the central limit theorem, a useful distribution of the error is the normal distribution. In the most general form of the theorem and under some conditions (which include finite variance) it states that averages of random variables independently drawn from independent distributions converge in distribution to the normal, that is, become normally distributed when the number of random variables is sufficiently large [29]. As shown in Figure 5, it is possible with the Harmonized Parabolic Synthesis method to accomplish the desirable normal distributed error.

V. IMPLEMENTATION DETAILS

The square root hardware is implemented in different hardware description languages such as Chisel, VHDL, and Verilog. The Chisel implementation is converted to Verilog to be synthesized via Xilinx tools. All implementations result in the same hardware and resource usage and support 3 out of 4 special input values defined by the IEEE binary32 format. The supported special input values are zero, ± infinity and, NaN (Not a Number). Denormalized numbers are currently not supported as we do not need them, however they can be easily supported in the future.

Two different implementations of the floating-point square root hardware are presented in this paper. The first implementation consists of a single stage and produces the output in 1 cycle, whereas the second implementation is pipelined and consists of 5 stages. The pipelined implementation produces the first output in 5 cycles and when the pipeline is filled 1 output per cycle is produced. Figure 6 shows the implementation with 5 stages. Each blue (relatively large) box corresponds to a step in the method. The vertical thick lines, which are separating the stages, represent the stage registers.

On chip DSP blocks, which have 27 × 18 input width in Ultrascale FPGAs and 25 × 18 on Virtex-7 FPGAs, are utilized for the multiplication operations. However, most of the multiplications are performed on inputs, which have longer bit-width than the width supported by the DSP blocks. Therefore, these operations are performed on more than a single DSP block that are cascaded.

The coefficient sizes are 28 bits, 22 bits, and 16 bits. Since 64 intervals are sufficient/used for performing square root on single-precision floating-point numbers, we need to store 64 values for each coefficient. The total amount of memory required for storing the coefficients become $64 \times (28 + 22 + 16) = 4224$ bits or 528 bytes.

VI. RESULTS

In this section, we provide post-implementation resource utilization and timing results of our square root hardware implementations and compare them to the results of several related works. The FPGA synthesis is performed by Xilinx tools. The target platforms are Zybo Zynq-7000 board with Virtex-7 series FPGA and Xilinx VCU108 evaluation kit including Virtex UltraScale XCUX095-2FFVA2104E FPGA. Zybo is a low-end board and VCU108 is a high-end board.

Two different implementations of the square root hardware with different number of stages have been evaluated. The first implementation is fully combinational (consisting of a single stage) whereas the second implementation is pipelined (divided into 5 stages). The implementations have different results in terms of clock frequency and resource usage. Maximum frequencies are measured using the out-of-context flow to synthesize and implement the methods in isolation. This ensures that the design is not distorted due to routing to device pins. These results are presented in Table I. In the combinational implementation no registers are used, however, in the pipelined implementation stage registers are used.
UltraScale FPGA has a higher technology than the Virtex-7 FPGA. Therefore the critical path becomes shorter and the clock frequency gets almost doubled while moving from Virtex-7 to UltraScale. Due to the differences in the configurable logic block (CLB) structures, the register and look-up-table (LUT) numbers vary between the FPGAs. The block ram structures are the same, thus the usage of BRAMs do not change. However, the DSP blocks in the UltraScale FPGA have slightly larger input width and this causes utilization of fewer DSP blocks.

When compared to Xilinx square root intellectual property (IP) [23], [30], which is implemented via CORDIC method [9], [10], our implementation on VCU108 uses fewer LUTs and FFs. However, the Xilinx IP achieves higher clock frequency due to not utilizing any BRAMs or DSPs and dividing the hardware into shorter pipeline stages. In total they have 25 stages, whereas we have only 5. In terms of latency, our implementation shows better performance due to having fewer pipeline stages. The latency of our implementation is less than half the latency of Xilinx IP. This comparison is given in Table II along with other studies.

The implementation of Dinechin et al. [22] uses 241 slices, which is approximately 1k LUTs and 1k FFs. They also make use of on board DSP units and block RAMs. The frequency they achieve is more or less the same when compared to our implementation on Ultrascale FPGA, however, their latency is more than double of our latency due to requiring more cycles to produce a result. Perhaps they could achieve higher frequency with a newer FPGA, however it is difficult to determine what the resource usage would be.

Suresh et al. [20] do not make use of the DSPs or BRAMs as they are using a non-restoring iterative method that produces a result in 40 cycles. The resource usage of their implementation is significantly higher than our implementations on both Virtex-7 and UltraScale FPGAs. The latency they get is approximately tenfold higher than our implementation on UltraScale and fivefold on Virtex-7. This is due to the large number of cycles they need for producing a result.

Hasnat et al. [24] uses 199 LUTs and 24 FFs which are quite low. However, we assume they use 14 DSPs which is above our numbers. Additionally, their latency is higher and throughput is lower than what we achieve.

We should note that applying a multiplicative method to compute the square root on an FPGA makes use of the on-chip DSP blocks, however, these blocks limit the maximum clock frequency by preventing the stages to be shorter. Especially if the inputs are too large, several DSP blocks can be concatenated and the stages can become even longer.

VII. CONCLUSION

We have presented a novel method for implementing single-precision floating-point square root operation based on combination of two methods namely parabolic synthesis and second degree interpolation. We have implemented single-cycle and pipelined versions of the method supporting the IEEE single-precision floating-point standard and synthesized on two different FPGA boards. We compared the resource utilization and timing results of our implementations with the results of other studies and a Xilinx IP.

We are aware that comparison between works with different FPGAs is not very accurate. We do the comparison to to give an opinion to the reader. Our implementations are smaller
### TABLE II
**COMPARISON OF OUR IMPLEMENTATION WITH SEVERAL OTHER IMPLEMENTATIONS.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>249</td>
<td>448</td>
<td>241 slices*</td>
<td>819</td>
<td>199</td>
</tr>
<tr>
<td>FF</td>
<td>172</td>
<td>786</td>
<td>241 slices*</td>
<td>531</td>
<td>24</td>
</tr>
<tr>
<td>DSP</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>BRAM</td>
<td>1.5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Frequency</td>
<td>240</td>
<td>568</td>
<td>237</td>
<td>205</td>
<td>199</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>5</td>
<td>25</td>
<td>12</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>Latency (ns)</td>
<td>20.8</td>
<td>44.0</td>
<td>50.6</td>
<td>195.1</td>
<td>60.3</td>
</tr>
<tr>
<td>Throughput</td>
<td>1 FLOP/cycle</td>
<td>1 FLOP/cycle</td>
<td>N/A</td>
<td>N/A</td>
<td>1 FLOP/12 cycles</td>
</tr>
</tbody>
</table>

than most of the other works with the exception of DSP and BRAM usage. In terms of latency, our designs outperform the other works. Therefore, they are suitable for high performance embedded systems. They can be used either standalone or integrated to another system. Both DSP and BRAM usages are open to optimizations as the data can easily fit into a single BRAM and number of the DSP blocks can be decreased by further optimizations on the sizes of intermediate results.

The error distribution is neglected in all studies. However, works based on CORDIC show an error distribution away from zero. Hence the errors can accumulate and cause accuracy problems in the final result. Our method is able to perform a normal distribution of the errors evenly around zero, which prevents accumulated error problems.

### REFERENCES


[7] E. Hertz, B. Svensson, and P. Nilsson, “Combining the parabolic syn-


[16] O. Kusheleva, “Babylonian method of computing the square root: Justi-


[23] (Xilinx Inc, 2017) Xilinx cordic v6.0 logi-


[30] (Xilinx Inc, 2017) Performance and resource uti-
Appendix F

Paper VI
A Configurable Two Dimensional Mesh Network-on-Chip Implementation in Chisel

Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström

submitted to 32nd IEEE International System-on-Chip Conference (SOCC), Sept. 3-6, 2019, Singapore.
A Configurable Two Dimensional Mesh Network-on-Chip Implementation in Chisel

Süleyman Savas, Zain Ul-Abdin and Tomas Nordström
School of Information Technology
Halmstad University
Halmstad, Sweden
suleyman.savas@hh.se, zain-ul-abdin@hh.se, tomas.nordstrom@hh.se

Abstract—On-chip communication plays a significant role in the performance of manycore architectures. Therefore, they require a proper on-chip communication infrastructure that can scale with the number of the cores. As a solution, network-on-chip structures have emerged and are being used.

This paper presents description of a two dimensional mesh network-on-chip router and a network interface, which are implemented in Chisel to be integrated to the rocket chip generator that generates RISC-V (rocket) cores. The router is implemented in VHDL as well and the two implementations are verified and compared.

Hardware resource usage and performance of different sized networks are analyzed. The implementations are synthesized for a Xilinx Ultrascale FPGA via Xilinx tools for the hardware resource usage and clock frequency results. The performance results including latency and throughput measurements with different traffic patterns, are collected with cycle accurate emulations.

The implementations in Chisel and VHDL do not differ much. Chisel requires around 10% fewer lines of code, however, the synthesis results are very similar. Our latency result are better than the majority of the other studies. The other results such as hardware usage, clock frequency, and throughput are competitive when compared to the related works.

Index Terms—network-on-chip, Chisel, mesh, scalable

I. INTRODUCTION

MultiProcessor System-on-Chip (MPSoC) designs have emerged due to performance requirements and thermal limitations. In order to increase the performance, these systems employ several processing elements (PEs), which can be general purpose processors, application specific processors, or accelerators. However, computation power is not the only factor that affects the performance of an MPSoC (or a manycore architecture). On-chip communication is another crucial factor especially for the architectures targeting dataflow applications. The large amount of data that is moved around the PEs plays a significant role in the performance. The usual infrastructure for on-chip communication is a bus system. However, this system does not scale well for high number of PEs. It does not support parallel communication and the wire delays increase with the number of PEs. Therefore, network-on-chip (NoC) infrastructure [1], that supports scalability, has become an accepted interconnect infrastructure in manycore architectures.

In this paper, we describe a configurable two dimensional mesh NoC router and a network interface (NI) to connect this router to a RISC-V based (rocket) core [2] that is generated with rocket chip generator [3]. The router implements packet switching with XY routing. Both the router and the NI are developed in Chisel and integrated into the rocket chip generator in order to generate manycore architectures with a scalable NoC. This task is performed as a part of a manycore design method described in a prior study [4]. The router has been developed in VHDL as well. We first give detailed description of the NoC router and the NI and then provide a brief comparison between Chisel and VHDL. Later, we analyze hardware resource usage and timing results of the Chisel implementations of the router, NI and network with different sizes synthesized on a Xilinx Ultrascale FPGA. Additionally, we measure the throughput and latency results of a 4x4 network implementation by injecting synthetic network packets into the network with different traffic patterns. Finally, we compare our results with the results of several related works.

II. REQUIREMENTS

The basic requirements of the NoC design is to support the memory operations of RISC-V. The basic memory operations are the load store instructions "LW/SW (LH/SH, LB/SB)" that transfer a value between the registers and memory; and the instruction reads. We can translate these instructions into the following NoC commands:

- Write request {8/16/32/64-bit}
- Read request {8/16/32/64-bit}
- Read reply {8/16/32/64-bit}

These NoC commands will consists of an address, the data, and a NoC control. Suitable sizes for these fields will be 64-bit data, 48-bit addresses, and then we set aside 16 bits for control to end up with 128 bits NoC commands as seen in Table I and II. For the read request the data field is used for the return address.

In addition to these required NoC commands, we think that it would be good if the NoC could support the following multiword operations. This could help cache operations as well as non-aligned LD/SD operations.

- Write req. multi {multiple 64-bit words, min 16 words}
- Read req. multi {multiple 64-bit words, min 16 words}
- Read reply multi not-final {64-bit}
- Read reply multi final {64-bit}

To implement these we can add fields to the NoC control, and only need to make minor changes to the router/NI.
A. Atomic operations

In addition to the load/store operations the RISC-V actually have an "A extension" with atomic instructions, which to the NoC will be special operations. The first set of atomic operations are the load-reserved (LR) and store-conditional (SC) instructions. Another class of atomic operations are the atomic memory operation (AMO) instructions which perform read-modify-write operations. The operations supported are swap, integer add, logical AND, logical OR, logical XOR, and signed and unsigned integer maximum and minimum.

We can translate these atomic instructions into the following NoC commands:

- Atomic request (LR, SC, RMW) {64-bit}
- Atomic reply (LR, SC, RMW) {64-bit}

To implement these atomic operations, we need to extend the NoC instruction with an additional address field. That is, these operations need two address and one data field (and control) in the NoC instruction, needing more than the target 128-bits. Alternatively, we can split these NoC instructions into two parts and have the receiver reassemble the instructions.

However, when using this NoC in a system realizing dataflow applications, there is no direct need for atomic operations, as all communication is implemented using FIFO communication. Thus, for this first NoC implementation we have not implemented support for atomic operations.

III. ROUTER AND NETWORK INTERFACE ARCHITECTURE

The network-on-chip (NoC) router is used to connect multiple CPU cores in a 2D mesh network inspired by the on-chip network of Epiphany architecture [5]. This network topology is highly regular, scales well (the cost of wires and network routers increase linearly with the number of cores), and can be made deadlock free with the XY routing protocol. The wire connections are to the nearest neighbour. Therefore, the wire delay does not increase with the number of routers. Additionally, since the connections are regular, they can be implemented with loops in the hardware description language.

The mesh consist of two physical, independent networks namely wMesh and rMesh. The write packets are sent over the wMesh, whereas the read packets are sent over the rMesh. The response to a read packet is sent over the wMesh as a write packet. There are no differences between the write packets and the read response packets from the router point of view. The router does not distinguish these packets from each other. Therefore, the separation is done by the network interface by using bit 78.

Figure 1 shows the internal structure of the router with the components for a single mesh. All mesh components (MeshRx and MeshTx) are duplicated for the other mesh. The CpuRx and CpuTx form the interface to the processing core, whereas the MeshRx and MeshTx form the interfaces to the neighbour NoC routers in each direction, for each mesh. The fifth MeshRx and MeshTx are the interfaces to the CpuRx and CpuTx.

The widths of the data buses within the meshes are configurable. A network packet is 128 bits and it needs to be divided into smaller chunks, which fit into the data buses, to be transferred on the mesh. The reason to use data buses narrower than 128 is to save resources and allow the mesh to run with a higher clock frequency than the CPUs. We use 32 bits for the current implementation. Each router has a separate FIFO buffer for each direction to store the incoming packets. The buffer size is configurable and it is set to 2 packets (256 bits) in the current implementation.

A. Network Packets and Routing

There are different network packets for wMesh (write) and rMesh (read) networks. Read packets are used for read request and the write packets are used for write requests and read replies. The packets on the wMesh consist of payload data and control signals whereas the packets on the rMesh do not have any payload data. The detailed view of the bit fields for the packets can be seen in table I and II.

**TABLE I: The bit fields of write packets on wMesh.**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>127 - 118</td>
<td>10 bits row address of the destination tile.</td>
</tr>
<tr>
<td>117 - 108</td>
<td>10 bits column address of the destination tile.</td>
</tr>
<tr>
<td>107 - 80</td>
<td>28 bits address to memory in destination tile.</td>
</tr>
<tr>
<td>79</td>
<td>1 bit for packet type. 1 = write packet.</td>
</tr>
<tr>
<td>78 - 64</td>
<td>15 bits for other control signals.</td>
</tr>
<tr>
<td>63 - 0</td>
<td>64 bits data</td>
</tr>
</tbody>
</table>

As seen in Table I, the write packets have 64 bits of capacity for the data. There are 16 bits in each packet for control signals. The router uses 10 bits for row addressing and 10 bits for column addressing to address the NoC routers. Thus, the network can support addressing up to 1,000,000 routers/tiles.

The network router implements the XY routing protocol while forwarding the network packets. This protocol is deadlock free if it follows the rules below:
TABLE II: The bit fields of read packets on \( \text{rMesh} \).

<table>
<thead>
<tr>
<th>Bits</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>127 - 118</td>
<td>10 bits row address of the destination tile.</td>
</tr>
<tr>
<td>117 - 108</td>
<td>10 bits column address of the destination tile.</td>
</tr>
<tr>
<td>107 - 80</td>
<td>28 bits address of memory in the destination tile.</td>
</tr>
<tr>
<td>79</td>
<td>1 bit for packet type; 0 = read packet</td>
</tr>
<tr>
<td>78 - 64</td>
<td>15 bits for other control signals.</td>
</tr>
<tr>
<td>63 - 48</td>
<td>not used.</td>
</tr>
<tr>
<td>47 - 38</td>
<td>10 bits row address of the return tile.</td>
</tr>
<tr>
<td>37 - 28</td>
<td>10 bits column address of the return tile.</td>
</tr>
<tr>
<td>27 - 0</td>
<td>28 bits return address in the return tile.</td>
</tr>
</tbody>
</table>

If the row and column addresses of the packet is to a router outside the mesh, the packet is sent to the upper left (north west) router, which has the lowest row and column addresses on the chip.

If the column address of the packet is less than the column address of the current router, the packet is sent to the left (west). If it is greater, the packet is sent to the right (east).

If the column and row addresses of the packet are equal to the column and row addresses of the current router, the packet is sent to the CPU.

If the column address of the packet is equal to the column address of the current router, and the row address of the packet is less than the row address of the router, the packet is sent upwards (north). If the row address is greater, the packet is sent downwards (south).

With the aforementioned rules, if a packet enters the router from the north or from the south, it will never be sent to the east or west. Therefore, the \( \text{MeshRx}s \) in the north and south, as seen in Figure 1, are connected to only two \( \text{MeshTx}s \). This does not change with respect to the location of the router in the mesh. However, the packets coming from east and west can be forwarded to any direction including the CPU.

B. CpuRx & CpuTx

Each router includes one CpuRx that is connected to the network interface (NI). This module reads the 128 bits from NI and stores in 128 bit buffers (separate buffers for \( \text{rMesh} \) and \( \text{wMesh} \)). The module sets a ready signal to indicate a buffer is empty and the NI can send a packet. It divides the 128 bit packet into smaller chunks based on the width of the data bus for \( \text{wMesh} \) and \( \text{rMesh} \) and sends to the corresponding \( \text{CPUMeshRx} \). The data bus widths for the current implementation are 32 bits for the meshes. The incoming packet is forwarded to the \( \text{wMesh} \) or \( \text{rMesh} \) based on bit 79, which determines the type of the packet (read/write).

CpuTx module is connected to the NI. It receives packets from the \( \text{rMesh} \) and \( \text{wMesh} \). It uses Round Robin scheduling to arbitrate between the meshes. It combines the small packets and creates the 128 bit packet to be sent to the CPU. It sets the valid signal to indicate the availability of the packet. The valid signal is kept set until the NI sets the ready signal.

C. MeshRx & MeshTx

There are 10 MeshRx modules in the network router. There is one MeshRx for each direction and one for the CPU on \( \text{rMesh} \) and \( \text{wMesh} \) individually. The routing is performed by this module. It receives the network packets either from the neighbour routers or the local CPU and forwards them to the target MeshTx module within the router. The FIFO buffers to store the incoming network packets are in MeshRx modules.

MeshTx module implements a Round Robin scheduler to receive the network packets coming from different directions. It forwards the packets either to the MeshRx module of neighbour routers or to the local CpuTx module.

D. Network Interface

We have developed a network interface (NI) to connect the network router to the rocket core. This NI is integrated into the rocket chip generator that generates full systems with rocket cores. With the NI and the router, the generator can generate manycore architectures with two dimensional mesh network. This NI is only for the RISC-V (rocket) core. However, any interface that supports the AMBA protocol can be connected to the NoC router.

The memory access protocol for the rocket core is not documented. Therefore we needed to do reverse engineering to figure out the protocol. The default local memory in the rocket chip generator is cache, however scratchpad is supported as well. The NI is placed between the core and the local memory, as illustrated in Figure 2, to have all memory accesses to go through the interface. The local accesses are forwarded to the local memory without any modification and the remote accesses are forwarded to the network. The NI uses Round Robin scheduling to arbitrate the communication between the core, memory and router.

![Fig. 2: The network router and the interface connected to a RISC-V (rocket) core.](image-url)
the response is un-cached) signals. The s1_kill signal is used to cancel the request of the previous cycle, the s1_data is the data to be written to the memory with the write request, and invalidate_lr is used to invalidate a cache line. The s2_nack indicates that the request that is sent two cycles ago is rejected, ordered signal is set if the request is accepted by the memory, replay_next is set when the response is uncached, and s2_xcpt carries the exception signals. Some of these signals such as the tag, typ and has_data are embedded into the control signals of the network packet and forwarded to the destination. The tag signal is 7 or 8 bits depending on the usage of the accelerator and it is used for keeping track of the destination registers for the read replies. The typ signal is 3 bits and used to determine the size of the data that is being read or written. The has_data signal is a single bit signal that indicates if the read reply packet includes data. The packet without data is simply an acknowledgment indicating that the read request is received and the data will be returned in a future reply.

Fig. 3: The signals between the rocket core and the network interface. Request and response consist of several signals each.

The rocket core sends the memory requests in two cycles. In the first cycle the control signals are sent and in the second cycle the data is sent. However, the NoC routers processes the control and data signals on the same cycle. Therefore the interface stores the control signals for a cycle and combines them with the data before forwarding them to the router. The connections between the router and the network interface can be seen in Figure 4. The network router uses ready and valid signals as in AMBA protocol [6] to communicate with the network interface. The width of data bus between the NI and the router is 128 bits, which is the size of a network packet.

Fig. 4: The signals between the interface and the router.

E. Chisel Vs. VHDL

We have implemented the NoC router in both Chisel and VHDL. In this section, we will compare these languages with code examples from the router implementations. The code in Listing 1 shows the I/O interface of the MeshTx module in VHDL and the code in Listing 2 shows the interface in Chisel. In VHDL, the I/O interface is an entity and the behaviour or structure is an architecture, which are implemented separately. However, in Chisel the I/O interface and the behaviour/structure are both implemented in a class that extends the Module class. The I/O interface of the module is defined with an IO() call that instantiates a Bundle with the I/O signals. Reset and clock signals are embedded into the IO bundle. The boolean type can be used to represent single bit signals in Chisel.

Listing 1: MeshTx interface in VHDL

```vhdl
entity MeshTx is
  generic ( gNumTransfers : Integer := 4;
            gNumDataBits : Integer := 32 );
  port ( RstN : in std_logic;
         Clk : in std_logic;
         to/from Mesh
           MeshD : out std_logic_vector(gNumDataBits-1 downto 0);
           MeshDValid : out std_logic;
           MeshDReady : in std_logic;
         to/from other Mesh interfaces
           NodeD : in tCNodeD;
           NodeDValid : in std_logic_vector(4 downto 0);
           NodeDReady : out std_logic_vector(4 downto 0) );
end MeshTx;
```

Listing 2: MeshTx interface in Chisel

```java
class MeshTx(val gNumTransfers: Int, val gNumDataBits: Int) extends Module with
  TypeConstants {
  val io = IO(new Bundle{
    // to/from Mesh
    val MeshD = Output(UInt(gNumDataBits.W))
    val MeshDValid = Output(Bool())
    val MeshDReady = Input(Bool())
    // to/from other Mesh interfaces
    val NodeD = Vec(5, UInt(INPUT, width = gNumDataBits.W))
    val NodeDValid = Input(UInt(5.W))
    val NodeDReady = Vec(5, Output(UInt(1.W)))
  })
```

The logic vectors of the VHDL implementation are implemented in two different ways in Chisel. The first one is the implementation of the MeshD signal. It is implemented as an output signal that is an unsigned integer with the width of gNumDataBits. The second way is the implementation of the NodeD and NodeDReady signals. They are implemented
as vectors of signals with different directions and sizes. Both implementations can represent a multi-bit signal. However, the first implementation does not support partial assignments such as `MeshD(2) = 1.U`, which tries to assign 1 to the third bit of the signal.

The Listings 3 and 4 present a variable sized random access memory (RAM) implementation in VHDL and Chisel, respectively. The `if` statements become `when` statements in Chisel. The `if` statements are supported in Chisel, however, they cannot have any hardware in the condition but only Scala variables [7]. In VHDL the RAM block is defined as a signal that is to be interpreted as a memory by the synthesizer. In Chisel the only thing to do is to instantiate the Mem module with the proper arguments (number of words, type and size of words). Reading and writing operations are similar.

```vhdl
entity DpRamAsyncRd is
generic(
  nAbits: integer := 4;
  nWidth: integer := 32);
port(
  DlnA: in std_logic_vector(nWidth−1 downto 0);
  WlnA: in std_logic;
  ClnK: in std_logic;
  AddrA: in std_logic_vector(nAbits−1 downto 0);
  AddrB: in std_logic_vector(nAbits−1 downto 0);
  DOutB: out std_logic_vector(nWidth−1 downto 0)
);
end DpRamAsyncRd;
architecture rtl of DpRamAsyncRd is
type tRam is array (0 to (2**nAbits−1)) of std_logic_vector(nWidth−1 downto 0);
signal Ram: tRam := (others => (others => '0'));
begint
  RamP: process (ClnK)
  begin
    if rising_edge(ClnK) then
      if WlnA = '1' then
        Ram(TO_INTEGER(unsigned(AddrA))) <= DlnA;
      end if;
    end if;
  end process;
  DOutB <= Ram(TO_INTEGER(unsigned(AddrB)));
end rtl;
```

Listing 3: RAM implementation in VHDL

```java
class DpRamAsyncRd(val nAbits: Int, val nWidth: Int) extends Module with TypeConstants{
  val io = IO(new Bundle{
    val ClnK = Input(Bool())
    val DlnA = Input(UInt(nAbits.W))
    val WlnA = Input(Bool())
    val AddrA = Input(UInt(nAbits.W))
    val AddrB = Input(UInt(nAbits.W))
    val DOutB = Output(UInt(nWidth.W))
  })
  val Ram = Mem((1 << nAbits), UInt(0, width = nWidth))
  when(io.WlnA){
    Ram(io.AddrA) := io.DlnA
  } else {
    io.DOutB := Ram(io.AddrB)
  }
```

Listing 4: RAM implementation in Chisel

The I/O interface implementations differ between VHDL and Chisel. We do not go into the details of the logic implementations. However, there are similarities, such as for loops and if/when statements and differences, such as memory and register implementations and register assignments. The source line of code, that is usually used for quantifying the productivity, for the implementations is 950 for Chisel and 1050 for VHDL. It should be noted that, for a developer who is not an expert in either of the languages, Chisel is easier to construct hardware. We have not used any functional programming features of the language. However, while implementing more complicated designs such as the rocket chip generator itself, we believe that for a developer who is familiar with functional programming and/or Scala, Chisel would be a more productive language. The differences between hardware resource usage and clock rate results are negligible. Thus, we will not present the results for the VHDL implementation.

IV. RESULTS AND ANALYSIS

In this section, we analyze the results of our network router and interface implementations in Chisel. We divide the results into two classes; performance and resource utilization and timing. The performance results, consisting of throughput and latency, are collected by running a 4 × 4 network on a cycle accurate emulator. Synthetic network packets are injected into the network with different patterns. The hardware results are collected with FPGA synthesis performed by Xilinx tools. The target platform is Xilinx VCU108 evaluation kit including Virtex UltraScale XCVU095-2FFV12104E FPGA.

Prior studies show that a direct low-diameter topology improves latency in NoCs [8], [9]. Therefore, we implement square shaped meshes. The hardware resource usage of the router depends on its location in the network. The different locations are corners, edges, and inside. We call these routers as corner routers, edge routers and central routers, respectively. Figure 5 illustrates these router types with a 3 × 3 network. As seen in the figure, the corner routers do not have any connections on two directions, the edge routers do not have any connections on one direction and the central router is connected in all directions. The transmitter and the receiver components, which are not connected, are removed automatically by the optimization of the FPGA tools. As a result, the corner routers become smaller than the edge routers and the edge routers become smaller than the central routers.

The resource usage and timing results for different router types, network interface and several mesh networks are given in Table III. When the network size increases, at the beginning, the resource usage does not increase linearly due to the dominance of edge and corner routers. However, the influence of these routers becomes smaller as the network size increases. Because the ratio of the number of the edge and corner routers over the total number of the routers decreases. The results show that with a 7 × 7 network, there would be approximately
7k LUTs for each core left on the utilized Ultrascale FPGA. This is enough for a tile with rocket core using no floating-point unit.

TABLE III: Timing and hardware resource usages results on a Xilinx Ultrascale FPGA.

<table>
<thead>
<tr>
<th>Component</th>
<th>LUT</th>
<th>FF</th>
<th>Clock Rate (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Interface</td>
<td>305</td>
<td>215</td>
<td>657</td>
</tr>
<tr>
<td>Corner Router</td>
<td>1588</td>
<td>1099</td>
<td>-</td>
</tr>
<tr>
<td>Edge Router</td>
<td>1772</td>
<td>1191</td>
<td>-</td>
</tr>
<tr>
<td>Central Router</td>
<td>2131</td>
<td>1285</td>
<td>460</td>
</tr>
<tr>
<td>1x2 Network</td>
<td>2373</td>
<td>2025</td>
<td>384</td>
</tr>
<tr>
<td>2x2 Network</td>
<td>6517</td>
<td>4401</td>
<td>370</td>
</tr>
<tr>
<td>3x3 Network</td>
<td>16,051</td>
<td>10,449</td>
<td>357</td>
</tr>
<tr>
<td>4x4 Network</td>
<td>29,880</td>
<td>19,069</td>
<td>344</td>
</tr>
<tr>
<td>8x8 Network</td>
<td>127,469</td>
<td>76,269</td>
<td>316</td>
</tr>
<tr>
<td>16x16 Network</td>
<td>522,011</td>
<td>323,125</td>
<td>250</td>
</tr>
<tr>
<td>Available</td>
<td>537,600</td>
<td>1,075,200</td>
<td>-</td>
</tr>
</tbody>
</table>

It takes 16 cycles for a core to send a (network packet) memory request to the memory of a neighbour (adjacent) core. The network routers contribute 15 cycles to this propagation delay and the remaining 1 cycle is added by the network interface. The cycles that is added by the interface is not included in the simulation results. It costs an additional 4 cycles per router if the packet is addressed to a tile that is not adjacent to the sender tile. The path that is taken by a remote memory request is shown in Figure 6. The CpuRx divides and forwards the packet that comes from the NI in 4 cycles to the IMesh or oMesh with the current configurations. During these 4 cycles, the router does not accept any new packet of the same type. Therefore, the packet frequency is 1 packet per 4 cycles on each mesh unless the network is busy due to traffic from other tiles. Ideally, the routers and the interface can run with a clock that has a higher rate than the clock of CPU cores in order to decrease the propagation delay in terms of CPU cycles. In case of running the NoC with a $4 \times 4$ clock rate, the delay will be 4 CPU cycles plus 1 CPU cycle for each additional hop. The same result would be achieved if the data widths between the routers were increased to 128 bits.

Figure 7 presents the throughput and latency analysis of a $4 \times 4$ mesh network consisting of only routers. We used three different traffic patterns to inject write request packets into the network. These patterns are bit-complement, uniform random, and hot point. In bit-complement, a router sends packets to the router that has bit-wise reverse address. In uniform random pattern the destination cores are randomly chosen and in hot point pattern all cores send packets to the same core. The injected packet size is 128 bits, however the packet is divided into 4 sub-packets (flits) of 32 bits when pushed into the mesh by the source router. Injections to the routers are done simultaneously i.e. 0.1 packet/cycle/router means all routers are injected with a packet at every 10th cycle.

The network shows a similar performance for bit-complement and uniform random patterns. In terms of throughput, the saturation point, which is around 1.3 packets/cycle (or 5.2 sub-packets (flits) /cycle), is reached when the packet injection rate is approximately 0.1 packet/cycle/router. The shortest latency (between adjacent routers) is 15 cycles. The average number of hops for a packet in a $4 \times 4$ network with XY routing is 2.63 with uniform random traffic and 4 with bit-complement pattern. This means 21.52 cycles of average latency for the first pattern and 27 cycles for the second pattern, ignoring any congestion. However, in practise, where the congestion plays a role, when the network traffic is at 0.05 packet/cycle/router the average latency increases to 25 cycles for the uniform random pattern and to 29 cycles for the bit-complement pattern. The average latency increases further when the injection rate is increased and saturates at 84 cycles for the bit-complement pattern and at 95 cycles for the uniform random pattern. The hot point pattern creates congestion around the destination core and the latency increases up to 900 cycles whereas the throughput decreases to 0.11 packet/cycle.
for the whole network.

V. RELATED WORKS

Retkowski and Göhringer [10] proposes a reconfigurable 2D mesh network-on-chip (RAR-NoC) with two different routing algorithms and a centralized monitor core. This core measures the channel utilization at run-time and the NoC is able to change the routing algorithm. They have developed interface to ARM and Microblaze processors. The router requires around 800 LUTs on a Xilinx Zynq FPGA. The latency per router is 2 cycles for header flit and 1 cycle for data flits with the west-first routing algorithm. The latency is 1 cycle for all flits with the XY routing algorithm. The flit sizes are configurable and set to 32 bits in their experiments. This means our packets, which are 4 flits, would have a latency of 4 cycles/router in this network. The delay per router is 4 cycles in our implementations as well (except the first and the last routers on the path, which take 5 and 10 cycles respectively).

Argo NoC [11] is a 2D mesh network with a novel NI that performs DMA and time division multiplexing (TDM) scheduling. It targets hard real time systems. The routers transfer the network packets without buffering, flow control or synchronization. A 4 × 4 ASIC implementation reaches 450 MHz of clock rate. Their NI, which uses around 700 LUTs and 800 FFs, is larger than ours. The worst case latency for 128 bit packets is reported as 148 cycles in a 4x4 bitorus network.

Catnap [12] is another 2D network employing concentrated mesh and multiple networks. It performs regional congestion detection to exploit power gating. The NoC we propose does also consist of multiple (two) networks. We use these networks to carry read and write packets separately, however, in Catnap, multiple networks are used for rerouting the packets regardless the packet type. They analyze the throughput and latency of a 256 core implementation with an 8 × 8 concentrated mesh topology (4 cores/router). The throughput for uniform random traffic increases linearly from 0 to 25 packets/cycle and saturates while the injection rate goes from 0 to 0.5 packets/router/cycle. The latency starts from 20 and increases exponentially to 80 cycles around 0.4 injection rate.

Heisswolf et al. [13] implement a 2D mesh network router that introduces a self optimization strategy called rerouting. The strategy focuses on reallocating existing guaranteed service (GS) connections, which are virtual channels established to provide quality of service (QoS), based on congestion. Their routers are much larger (24k LUTs, 3.6k FFs) and achieve lower clock rate (100MHz) when compared to ours. They implement a 10 × 10 network for latency and throughput analysis. Their average latency is around 80 cycles for 0.05 flits/cycle/router and it increases to 135 for 0.3 flits/cycle/router. Our results show a lower latency per packet that consists of 4 flits.

Grot et al. [14] implement a 2D mesh NoC, called KiloNoC, with a topology-aware QoS architecture to increase the efficiency of area, energy, and performance. They implement a few different versions and provide ASIC area results and simulation results for average packet latency using the uniform random traffic pattern. Their best implementation shows latency of 20 cycles when the injection rate is at 0.01 packet/cycle/router and it increases exponentially to 60 cycles at around 0.25 injection rate. The throughput results are relative to a prior work of theirs, thus we will not go through them.

Kwon and Krishna [15] proposes a NoC generator framework (OpenSMART) that is implemented both in Bluespec System Verilog [16] and Chisel. Similar to OpenSMART, OpenSoC Fabric [17] is a NoC simulator implemented in Chisel. They support generation of C++ emulator and synthesizable Verilog as do we. However, one needs the NoC router itself in Chisel to integrate it to the rocket chip generator. Therefore, we implemented the router itself in Chisel instead of implementing a framework to generate the NoC implementation in other languages.

There are studies on NIs such as [18]–[20], however, there is no NI that connects rocket cores, which are generated by the rocket chip generator, to any scalable NoC. Additionally, there is no NoC implementation that is written directly in Chisel and can be integrated into the rocket chip generator. It should be noted that the latency and throughput numbers are extracted from the figures in the corresponding papers.

VI. CONCLUSIONS

We presented a 2D mesh NoC router together with an interface to the rocket core that is based on RISC-V instruction set architecture and generated with rocket chip generator. We integrated the network interface and the router into the rocket chip generator in order to support generation of manycore architectures with a scalable NoC. We generated and synthesized networks with different sizes on a Xilinx Ultrascale FPGA. Additionally, we measured the throughput and latency results of a 4 × 4 mesh network with different traffic patterns and analyzed these results. We compared the Chisel and VHDL implementations in terms of productivity, hardware resource usage and clock rate.

The Chisel implementation has approximately 10% fewer lines of code, however, the generated Verilog implementation shows the same result in terms of hardware resource usage and clock rate as the VHDL implementation when synthesized. The resource usage of the implemented router is competitive when compared to related works. Our latency and throughput results are competitive as well. The latency results seem to be better than the other results, however, the throughput results are not as good. These results can be improved with wider data paths between routers. Furthermore, the routing mechanism can be simpler if the 128 bit packet is not needed to be divided to fit into the data paths. Finally, we can say that the results are satisfactory for a first implementation of a mesh NoC for the rocket chip generator.

REFERENCES

Fig. 7: Throughput and latency analysis of a $4 \times 4$ mesh network by injection of synthetic (128 bit) network packets with different traffic patterns.


Appendix G

Paper VII

A Framework to Generate Domain-Specific Manycore Architectures from Dataflow Programs

Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström

*to be submitted to Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), 2019.*
A Framework to Generate Domain-Specific Manycore Architectures from Dataflow Programs

Süleyman Savas, Zain Ul-Abdin and Tomas Nordström

School of Information Technology, Halmstad University
Halmstad, Sweden

Abstract
In the last 15 years we have seen, as a response to power and thermal limits for current chip technologies, an explosion in the use of multiple and even many computer cores on a single chip. But now, to further improve performance and energy efficiency, when there are potentially hundreds of computing cores on a chip, we see a need for a specialization of individual cores and the development of heterogeneous manycore computer architectures.

However, developing such heterogeneous architectures is a significant challenge. Therefore, we propose a design method to generate domain specific manycore architectures based on RISC-V instruction set architecture and automate the main steps of this method with software tools. The design method allows generation of manycore architectures with different configurations including core augmentation through instruction extensions and custom accelerators. The method starts from developing applications in a high-level dataflow language and ends by generating synthesizable Verilog code and cycle accurate emulator for the generated architecture.

We evaluate the design method and the software tools by generating several architectures specialized for two different applications and measure their performance and hardware resource usages. Our results show that the design method can be used to generate specialized manycore architectures targeting applications from different domains. The specialized architectures show at least 3 to 4 times better performance than the general purpose counterparts. In certain cases, replacing general purpose components with specialized components saves hardware resources. Automating the method increases the speed of architecture development and facilitates the design space exploration of manycore architectures.

Preprint to be submitted to Microprocessors and Microsystems
1. Introduction

Applications of today such as audio/vision processing, wireless communication and machine learning, process massive amount of data produced by all kinds of sensors. These applications require high computation power in order to provide results in a reasonable amount of time. The demand for higher performance has always been there throughout the history of the microprocessors. One major solution to the continuous demand for higher computation power has been the technology development. It enables higher chip densities and higher clock rates to provide higher computation power. As a consequence of higher density and clock rates, the power consumption increases, which causes higher chip temperatures. However there are thermal limitations to the chip temperature [1]. These limitations have led the industry to processors with lower clock rates and higher numbers of cores, which can run in parallel. Eventually, parallelism has become the key to greater performance.

Initially, multicore/manycore architectures were produced by using several identical cores on the same die. As the design space is explored, it is realized that heterogeneous architectures, consisting of different(specialized) types of cores, have the potential to provide higher performance than the homogeneous counterparts [2, 3, 4, 5]. Our results acknowledge this statement.

The complexity of the architectures grow as the performance and power requirements push them towards parallelism and heterogeneity. As a result, the difficulty of designing and programming these architectures increases. The hardware architects need to design each specialized core separately instead of creating identical copies of a core. Memory coherence, accesses and on-chip communications might also become more complicated due to having cores with different characteristics [5]. On the software side, the architects need to know all the details about parallel programming and the target architecture to be able to develop efficient applications.

In addition to the challenges in designing heterogeneous manycore architectures, verification and manufacturing costs are significant obstacles in exploring the design space of these architectures. There are simulators such as Gem5 [6] and SimFlex [7], which provide full system simulations to overcome these obstacles and evaluate architectures with different configurations.
However, based on the system size, the simulation times can be dramatically long [8]. Additionally, with the simulators, one cannot collect data on hardware resource usage and the maximum clock rate that can be achieved by the architecture.

In a prior study [9], we proposed a design method to develop specialized single core architectures with RISC-V cores directly from dataflow applications. The method consisted of 4 steps, however, only the first 3 steps were implemented. In the first step the application is developed in a high level dataflow language (CAL actor language [10]). This application is fed to software tools (TURNUS[11] and Cal2Many[12]) in the second step where the compute intensive parts (hot-spots) of the application are identified and converted into hardware accelerators implemented in Chisel language[13]. This language is a hardware description language based on Scala. The rest of the application is converted into software code that can be compiled for the target architecture. In the third step, the hardware accelerators are integrated to a RISC-V core that is configurable and a single core system is generated with another tool (rocket chip generator[14]). We developed a hardware library for basic floating-point operations and complex number arithmetic as well as some software tools and combined them with existing open source tools to automate the steps of the design method. In this study we complete the design method by implementing the fourth step, which is connecting multiple heterogeneous cores to each other with a 2 dimensional mesh network-on-chip (NoC). That is, the design method still starts from developing the application in CAL actor language and now ends with generating manycore architectures consisting of a 2D mesh NoC and RISC-V cores with scratchpad memories and different accelerators in the form of instruction extensions targeting hot-spots of the application. The NoC router is developed in VHDL and Chisel languages separately. The details and comparison of the implementations, performance analysis, area usage on FPGA, and comparison to other studies are given in [15]. In this study, we extend the rocket chip generator by integrating this router. We develop a new hardware/software code generation back-end to support generation of parallel C code and multiple accelerators. Additionally, we add a communication mechanism to the software generation to support core-to-core communication. Finally, we evaluate the complete design method.

We used two case studies in order to test and evaluate the design method, the tools, and the generated hardware & software implementations. We generated architectures with different configurations (targeting the chosen ap-
applications) within these case studies. The applications are executed on cycle accurate, full system emulators generated by the rocket chip generator to collect the performance results. The rocket chip generator generates synthesizable verilog code as well. We synthesize this verilog code on a Xilinx Ultrascale FPGA to collect area and timing results.

The first case study implements the autofocus criterion calculation, which is a key component of synthetic aperture radar systems [16]. In our previous work, we have implemented this case study in a sequential manner and executed on a single core. In this study, we implement two different parallel versions of it and run on 2 cores and 13 cores separately. The performance results of this study are compared to the results from [17, 16] in which the application is executed on commercial architectures. The second case study is the first convolution layer of GoogLeNet [18] as a proof of concept to demonstrate that the design method can be used for generating architectures targeting the machine learning domain. This case study is first executed on a single core architecture, then on 5 and 4 core architectures with accelerators. The other layers can be mapped on additional cores to have a manycore architecture that can execute the whole application. We used on-chip memory to store the data for the first layer, however, the memory requirements of the whole application will lead to the use of external memory, which might harm the overall performance.

The contributions of this study can be summarized as:

- The design method that is proposed in our previous study [9] is extended to generate manycore architectures by integration of a two dimensional mesh NoC router to the tool chain.

- The hardware library that is used while generating hardware accelerators is extended with a hardware block that performs 7x7 convolution.

- A new back-end for hardware & software code generation is developed within the Cal2Many framework, in order to support
  - parallel code generation for multiple cores
  - generation of multiple accelerators per core
  - on-chip communication
  - placement of code and data on the local memory of the corresponding cores
large data transfers to the accelerators via direct memory connections

- The tools and the generated architectures are evaluated with two different case studies from different domains.

- The performance of the generated architectures are compared with the performance of commercial architectures.

The rest of the paper is structured as follows: In Section 2 the related works are presented. Section 3 describes the step-by-step realization of the design method. Section 4 provides description of the case studies and details of the implementations. The results of the case studies follow in Section 5. In the same section, the results are discussed. Finally in Section 6, the study is concluded.

There has been a significant amount of work on simulating single core and multicore processors [6, 19, 20, 21, 22, 23]. However, simulation is slow [8] and provides neither area and timing results nor a realizable/synthesizable output. There are a limited number of studies providing these results and the outputs with different frameworks.

McPat[24] is a framework that models power, area and timing of multicore/manycore processors. This framework supports different cache models, NoCs and cores available in 2009 such as Sun Niagara and Intel Nehalem. The framework does not provide performance results but an XML interface for the performance tools. It does not provide any register-transfer-level (RTL) implementation or any other synthesizable output.

OpenPiton [25] is an open source framework for building scalable architectures from single core to manycores. It utilizes SPARC v9 ISA with a distributed cache coherence protocol and three 2D mesh NoCs. The framework supports FPGA and ASIC synthesis however, in contrast to the tools in our design method, there is no support for scratchpads or instruction extensions (custom hardware accelerators) and the configuration for the cache sizes is limited to two.

OpenSoC System Architect [26] is another framework that aims to build scalable architectures and synthesize them on ASICs or FPGAs. This framework utilizes Chisel language and rocket chip generator and supports RISC-V cores and instruction extensions. Additionally, it uses OpenSoC Fabric system-on-chip interconnection and routing framework [27] to generate
manycore architectures. The framework provides testbenches and verification tools. Our design method and this framework both use the rocket chip generator tool and therefore have similarities in case of the generated cores. Both frameworks generate rocket cores [28] and support instruction extensions and all the configurations supported by the rocket chip generator. However, our design method starts by application development in a high level language that increases the abstraction level and facilitates the application development. Additionally it includes application profiling for hot-spot identification and automatically generating the custom hardware accelerators whereas in the OpenSoC framework the accelerators must be identified and implemented manually by the developer.

Opencelerity [29] is an accelerator-centric system-on-chip (SoC) design based on a combination of 3 tiers namely general purpose, manycore and specialization tiers. This SoC uses rocket cores and RoCC [30] interfaces as in the architectures generated by our design tools. The first tier consists of 5 rocket cores capable of running Linux, whereas 506 smaller RISC-V cores reside in tier 2, and accelerators in tier 3. The accelerators are generated using SystemC and high-level synthesis tools. The accelerators and the cores are placed in different tiers and all cores share the accelerator tier. In contrast, we generate the accelerators automatically from the CAL language and each core can have its own tightly-coupled accelerators, making the accelerator an instruction extension to the core. Additionally, our design method starts with application development and uses application requirements to configure the architecture in terms of the number of cores, memory size, accelerator types, however, Opencelerity is a single architecture with a configurable accelerator tier.

Hussain et al. [31] proposes an architecture with a RISC core and multiple different accelerators similar to Opencelerity. However, they use a single core and connect it to the accelerators with a network-on-chip. This allows the accelerators to have access to each other and to the RISC core, which means an accelerator can be shared. Our current tools do not allow connecting an accelerator to the network but to the core directly. Therefore, the accelerators are not shared. We see the impact of this fact in our second case study. However, this support can be added to the rocket chip generator by extending it.
2. Design Method

In order to find the most suitable architecture for the application in hand, one needs to execute the application on different architectures and compare them. Instead of going through this effort, we propose developing the suitable architecture for the application by augmenting base cores with custom accelerators targeting hot-spots of the application. However, developing an architecture manually from scratch is a great challenge. Therefore, we use open source rocket chip generator for generating the architecture. We integrate the accelerators and a 2D mesh NoC to this generator to support generation of application specific manycore architectures. This process is divided into 4 main steps as follows:

1. Application development
2. Analysis and code generation
3. Accelerator integration
4. System integration

The steps of the realization of the design method are illustrated in Figure 1 together with the tools and their inputs and outputs. The generic description of all the steps and the realization of the first 3 steps are given in our previous study [9]. Still, we will give a summary of these steps together with the changes and additions we have done within this study and focus on the realization of the system integration step.

2.1. Application Development

In the application development step the developer writes the application code to be fed to the analysis and code generation tools. In the realization of this step we chose the CAL actor language [10] due to its support for parallelism, simplicity, and suitable software tools. A CAL program consists of actors, which are stateful operators. Actors have ports to communicate with each other and actions for computations. The actions are the code blocks that can change the state of the actor, consume inputs, perform computation and produce outputs. The actions can be fired based on the state of the actor and availability or value of the input data. Actors may build a network when several of them are connected to each other.

The number of cores in the generated architecture is determined based on the number of the actors implemented in this step. In certain applications the compute intensive parts may not be obvious. The developer can combine
or split certain actions to have a visible compute intensive action that can be converted into hardware accelerator.

2.2. Analysis and Code Generation

In the analysis and code generation step we utilize two different software tools namely TURNUS [11] and Cal2Many [12] for profiling and code generation respectively. TURNUS analyzes the CAL application and provides information such as firing rates of the actions, the number of executed operations, the number of tokens consumed and produced, and communication buffer utilization. It allows setting weights manually for each operation (both memory and arithmetic). In our case studies, we use the number of cycles...
taken by each operation on the target (rocket) core as the operation’s weight. By combining the weights with the firing rates of the actions and the number of the executed operations, TURNUS provides a bottleneck analysis of the CAL application with respect to the model of the target core. In this analysis, compute intensive (or bottleneck) actions are exposed. Figure 2 presents the bottleneck analysis for the convolution case study. The weight given in the figure is a product of operation weights, number of executed operations and number of the firings. It is clearly visible that the applyFilter action is the compute intensive part (or the bottleneck) of the actor with almost 50% of all the weights.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Action</th>
<th>Firings</th>
<th>Overall</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>applyFilter</td>
<td>2408448</td>
<td>42.98%</td>
<td>2895419520.00</td>
</tr>
<tr>
<td>conv1</td>
<td>conv1</td>
<td>802816</td>
<td>14.29%</td>
<td>893534208.00</td>
</tr>
<tr>
<td>conv1</td>
<td>conv1</td>
<td>802816</td>
<td>14.29%</td>
<td>893534208.60</td>
</tr>
<tr>
<td>conv1</td>
<td>conv1</td>
<td>802816</td>
<td>14.29%</td>
<td>891125760.00</td>
</tr>
<tr>
<td>conv1</td>
<td>accumulate</td>
<td>802816</td>
<td>14.29%</td>
<td>44219840.00</td>
</tr>
</tbody>
</table>

Figure 2: Bottleneck analysis of an actor with 5 actions, performing the first layer of the GoogLeNet (provided by TURNUS).

Once the compute intensive action(s) of the application is identified, a prefix \texttt{\_acc\_} needs to be added to its name for it to be recognized by the code generation tool, Cal2Many. This tool takes a CAL application and generates target specific code depending on the chosen back-end. There are several back-ends generating sequential \texttt{C}, parallel \texttt{C} for Epiphany architecture [32], \texttt{aJava} and \texttt{aStruct} languages for Ambric architecture [33], \texttt{Scala} subset for ePuma architecture [34], and a combination of \texttt{C} and \texttt{Chisel} languages for single rocket core architectures with accelerators. Within this study we have developed a new back-end to support combination of parallel \texttt{C} and \texttt{Chisel} languages for architectures with many rocket cores that can have a single or multiple accelerators. The actions with the \texttt{\_acc\_} prefix in their names are converted into hardware blocks with the help of a hardware library in \texttt{Chisel} language. The hardware library consists of basic floating-point operations, complex operations, and 7x7 convolution that is added within this study. The rest of the application is converted to \texttt{C} that can be compiled for the RISC-V core. The \texttt{C} code includes a communication mechanism for core-to-core communication and custom instruction calls to interact with the
accelerator(s).

The new back-end generates a C and a header file for each actor along with a single main C file. All of these files are compiled and linked together. The main file includes a function to synchronize the cores and another function to call the entry (scheduler) functions of each actor to be executed on the corresponding core based on the core id. The latter function for a 5 core implementation is presented in Listing 1. (splitter, convR, convG, convB, and combiner are separate CAL actors converted into C files.) This function is where the mapping of actors onto cores is performed. Each actor must be mapped onto the core that is generated for it. For instance, the convR() function, that is seen in the listing, makes use of an accelerator. Therefore it must be executed by a core that has the accelerator.

```c
void thread_entry (int cid, int nc) {
    syncCores(cid, nc); // synchronize the cores
    switch (cid) {
        case 0:
            splitter();
            break;
        case 1:
            convR();
            break;
        case 2:
            convG();
            break;
        case 3:
            convB();
            break;
        case 4:
            combiner();
            break;
        default:
            break;
    }
}
```

Listing 1: The main function that is executed on all cores.

The code examples in Listing 2, 3 and 4 show an example of converting a CAL action into C code with and without using accelerators. Listing 2 presents the original CAL action to be converted into C. (This action differs from the one in Figure 11.) The C code in Listing 3 makes use of the accelerator that performs the computations (applying the filter to an image). The C
code copies the input data into the source registers of the custom instruction that is forwarded to the accelerator, uses a macro to execute this instruction, and copies the result from the accelerator into a global variable. Listing 4 shows the version of the C code that does not use an accelerator but perform the operations locally. Both in CAL and in C code, there are global variables declared outside the scope of the code snippets.

```c
applyFilter : action =>
  var
    int i := 0,
    float result := 0.0,
    int j := 0,
    float tmp
  do
    while i < filterRowSize do
      j := 0;
      while j < filterColSize do
        tmp := filters[filterCounter][i][j] * patch[i][j];
        result := result + tmp;
        j := j + 1;
      end
      i := i + 1;
    end
  outputR := result;
end
```

Listing 2: An example CAL action that applies a filter to a patch of an image.

```c
static inline void conv1Inst__acc__applyFilter ( ) {
  uint64_t acc_input0;
  //Copy the inputs into 64 bit registers
  memcpy(((char*)acc_input0) + 4, &conv1Inst_filterCounter, 4);
  uint32_t tmpInput1 = ((char*)conv1Inst_patch) + 0;
  memcpy(acc_input0, &tmpInput1, 4);
  uint64_t outputReg;
  //Lets execute the custom instructions
  ROCC_INSTRUCTION(XCUSTOMACC, outputReg, acc_input0, 0,
                   FUNCT_FIRE);
  //Lets read the result
  memcpy(&conv1Inst_outputR, &outputReg, 4);
}
```

Listing 3: The generated C code that uses accelerator.
static inline void conv1InstapplyFilter ( ) {
    uint64_t outputReg;

    for (int i = 0; i < 7; i++)
        for (int j = 0; j < 7; j++)
            outputReg += conv1Inst_patch[i][j] * filter[i][j];

    conv1Inst_outputR = outputReg;
}

Listing 4: The generated C code that does not use any accelerator.

The core-to-core communication mechanism is implemented with flags (ready & valid) and FIFO buffers to create unidirectional virtual channels between the connected cores. Figure 3 illustrates two cores with two virtual channels directed to opposite directions. Core1 sends data to core2 through channel-1 whereas core2 sends data to core1 through channel-2. One such channel is generated for each connection between the CAL actors. The mechanism utilizes only write accesses through the network-on-chip and read accesses only to the local memory. This is due to the read accesses being more expensive than the write accesses on the NoC in terms of clock cycles. Initially, supported buffer size is 1 word that can be of any variable type. The current buffer size is enough for our case studies, however, future applications may require larger buffers. Then, the communication mechanism can be extended. We used generic communication mechanisms [35, 36] that support variable buffer sizes with the Epiphany architecture in prior studies [37, 38]. However, these mechanisms added a significant overhead to the communication time, and consequently to the overall execution time. Therefore, we keep the communication mechanism, that is developed within this study, as simple as possible.

Figure 3: Two cores connected to each other with two unidirectional virtual channels.
During the code generation each actor is mapped onto a different core making the number of cores equal to the number of actors. However, the Cal2Many framework supports split and combination of the CAL actors. Thus, the actors do not need to be mapped on individual cores. The new hybrid back-end does not make use of this feature yet. However, for the future applications, this feature can be embedded into the new back-end. The data and the code are placed in the local memory of the mapped core with the help of section attribute of the compiler (gcc for RISC-V). The linker description must define the necessary sections, which are used with the section attribute. The code snippet in Listing 5 shows how the section attribute is used with a label that is defined in the linker description, given in Listing 6.

```c
1 uint64_t * convG_ch1_buffer SECTION(".core1.data");
2 volatile int * convG_ch1_ready SECTION(".core1.data");
3 volatile int * convG_ch1_valid SECTION(".core1.data");
```

Listing 5: Variable declarations within the memory of core1.

```c
1 . = 0x80040000;
2 _core1_start = .;
3 .core1 : {
4   *(.core1.text)
5   *(.core1.data)
6   _core1_data_end = .;
7 }
```

Listing 6: Address setting for the memory of core1 in the linker description.

2.3. Accelerator Integration

We use the rocket custom co-processor (RoCC) [30] interface within the rocket chip generator to connect the generated hardware accelerators to the rocket core that executes RISC-V instruction set. The rocket core is an inorder scalar processor featuring a five-stage pipeline and an integer ALU. The rocket chip generator can generate systems-on-chip with different configurations such as number of cores, availability of the FPU, availability of accelerators, data memory type (cache or scratchpad), and size of data and instruction memories. The accelerators are integrated into the rocket chip generator where they can be instantiated and bound to a core.

The integration of an accelerator is shown in Listing 7 with a simple example. The generated accelerator is a Chisel module called acc. This module is instantiated in a class (AccModule) that extends the RoCC interface (LazyRoCCModule). The connections between the core and the accelerator are
established in the `AccModule` class, through the `io` bundle that consists of several signals for requests from the core and responses from the accelerator.

```scala
class AccModule(outer: ConvAcc)(implicit p: Parameters) extends LazyRoCCModule(outer) with HasCoreParameters{
  val accInst = Module(new acc())
  val cmd = Queue(io.cmd)

  accInst.io.in1 := cmd.bits.rs1(63, 32)
  accInst.io.in2 := cmd.bits.rs1(31, 0)

  io.resp.valid := accInst.io.valid
  io.resp.bits.rd := cmd.bits.inst.rd
  io.resp.bits.data := accInst.io.result
  cmd.ready := true.B
}
```

Listing 7: Integrating the accelerator into the rocket chip generator by extending the RoCC interface.

The instantiation of the `AccModule` and binding to a custom instruction is performed during instantiation and configuration of a core in the system configurations of the rocket chip generator. The code example in Listing 8 shows an example of instantiating an accelerator that is bound to custom instruction 0.

```scala
rocc = Seq(RoCCParams(
  opcodes = OpcodeSet.custom0,
  generator = (p: Parameters) => {
    val Acc = LazyModule(new AccModule(p))
    Acc
  }
))
```

Listing 8: The instantiation of an accelerator that is bound to a custom instruction.

Through the RoCC interface the hardware accelerators become an extension to the instruction set. The accelerators can be fired with a single instruction call. Based on a bit field in the instruction, the core may or may not halt until the accelerator returns a result. In our implementations we halt the cores while the accelerators perform a computation.

The accelerators have direct access to the local data memory through the RoCC interface. In our previous study [9] there was no need to transfer large amounts of data to the accelerators. Therefore the memory connections were not used. However, in this study, the convolution case study requires
large amounts of data to be transferred to the accelerator (in the form of arrays in the software code). In order to support the large data transfer we modified the code generation. Previously, all of the data would be transferred with instruction calls (by providing the source registers in the instruction). Now our back-end supports two types of data transfer. If the data to be transferred is not an array it is still transferred through instructions calls. However, if the data is an array, the address and the size of the array are transferred to the accelerator with a single instruction call.

The code snippet in Listing 9 shows how the accesses to the memory are performed within the extended RoCC interface. The accesses can be performed in the accelerator as well, if the io signal are forwarded to the accelerator. The upper part of the code shows a read request to the memory and the lower part shows how the response from the memory is read.

```scala
// Send read request to the memory
when(readMem) {
    io.mem.req.valid := true.B
    io.mem.req.bits.addr := dataAddress //
    io.mem.req.bits.cmd := MXRD // perform a load (MXWR for stores)
    io.mem.req.bits.typ := MTW // data size, D = 8 bytes, W = 4, H = 2, B = 1
    io.mem.req.bits.data := Bits(0) // we’re not performing any stores
}

// Read the response from the memory
when(io.mem.resp.valid) {
    convAcc.io.conv1Inst_inputImage_in := io.mem.resp.bits.data
}
```

Listing 9: Memory connections in the extended RoCC interface.

It is required to create a Chisel class in rocket chip generator that extends the RoCC interface to instantiate the accelerator and connect the accelerator to the interface. However, if necessary, it can be extended to read the instruction and the source registers from the core, store the input data, access the memory, provide inputs to the accelerator, fire the accelerator and read the outputs. The task of extending the RoCC interface is not automated yet. This task needs to be done once for each accelerator. There is no limitation
on the number of classes extending the interface. These classes can be re-
used to instantiate and connect the accelerator to a core while generating an
architecture.

2.4. System Integration

The result of the first three steps of the design method is application spe-
cific tiles with core and accelerators. In this fourth step, we connect the tiles
with a network-on-chip to generate application/domain specific manycore
architectures.

2.4.1. Network-on-Chip

The rocket chip generator supports a crossbar network by default. How-
ever, this topology does not scale well [39], especially when the number of
the cores reach to hundreds. Since every core is connected to every other
core, the cost of wires and routing components increase exponentially when
the number of cores is increased. Additionally, the length of wires may con-
tribute to the critical path and decrease the maximum clock frequency of
the architecture. Therefore, we implemented a new scalable network-on-chip
[15] in Chisel and integrated it to the rocket chip generator. Inspired by
the Epiphany architecture, which we evaluated in our prior studies [37, 38],
we chose the two dimensional mesh network-on-chip with XY routing for the
on-chip communication. This network topology is highly regular, scales well
(the cost of wires and network routers increases linearly with the number of
cores), and can be made deadlock free with the XY routing protocol [39].
The wire connections are to the nearest neighbour. Therefore, the wire delay
does not increase with the number of routers. Additionally, since the con-
nections are regular, they can be implemented with loops in the hardware
description language. In this topology, the same router is instantiated for all
network sizes. The routing decision is distributed, hence there is no central
decision point that could be a bottleneck.

2.4.2. Integration of the NoC to the Rocket Chip Generator

By default, the rocket chip generator generates separate instruction and
data caches for the rocket core. The core is connected to a memory module
that takes care of routing the memory accesses to the local memory (data
cache) or to the other components through the crossbar network. The address
space is shared by all the cores and other components in the architecture.
The default local memory type is cache, however for the data storage we
use scratchpad memories in order to avoid dealing with cache coherence, which is highly challenging to maintain while dealing with many cores [40]. Additionally, scratchpads require a simpler interface to the network due to being simpler than the caches. The instruction cache remains untouched as it is not possible to replace or remove it from the rocket chip generator. We name the combination of core, memory and/or accelerator as ‘tile’ as seen in Figure 5.

In order to integrate the mesh network to the rocket chip generator, we performed the following steps including the modifications to the rocket chip generator:

1. Implementation of our NoC router in Chisel.
2. Implementation of an interface between the core and the memory. This interface forwards the local memory accesses to the memory module and the global memory accesses to the router. The interface can be seen as the ‘NoC Interface’ in Figure 5.
3. Generation of a router per tile if the number of tiles is greater than 1.
4. Generation of connections between tiles and routers. The routers get connected to the interfaces developed in the second step.
5. Generation of connections between the routers.

The details of the first step are given in [15]. The connections between the core and the memory module are defined in a class called HellaCacheIO. This class consists of request, response, interrupt, and additional control signals. The network interface uses this class of signals to communicate with both the core and the memory. The connections between the core and the memory (utilizing the HellaCacheIO signals) are shown in Figure 4. The request signals are address, data, request type, data size, and return register. The response signals cover the request signals together with a few additional signals to indicate if the response has data, if the response is after a cache miss, etc.

The interface controls the address data on each memory request from the core to forward the request either to the local memory or to the network. If the request is to the local memory, the signals from the core are directly forwarded to the memory module, without any modification. However, if the request is to a remote memory, the request signals are converted into a network packet and forwarded to the router. If the request is to an address that is not a local memory of any core, then the request is forwarded to a specific core (core0 in the current implementation). On the receiving side,
the network interface receives the request from the router and converts it into the HellaCacheIO signals and send to the memory. The network interface requires the dimensions of the mesh and the size of the local memories (scratchpads) to translate the address into coordinates and prepare the network packets. The interface arbitrates the memory access between the core and the router in a Round-Robin fashion. The same method is used (between the core and the memory) for accesses to the router. The signals between the interface and the router consist of ready (1 bit), valid (1 bit), and data (128 bits) signals. The data signal includes the control signals as well as the actual data. If an interface receives a request from the router and the local memory is not ready yet, it stores the request until the memory is ready. Similarly, if the local memory returns a response to a request, that came from a remote core, and the router is not ready, the response is stored in the interface until the router becomes ready.

A write request from core to the memory is sent in two cycles. During the first cycles, the control signals are sent and during the next cycle the data is sent. However, the NoC router requires all of the signals in one cycle. Therefore the interface stores the control signals for a cycle and combines with the data before sending to the router. This raises an issue because the
NoC router accepts one request at every fifth cycle, which means either the core needs to check if the router is ready to send a request or simply do not send another request for at least 4 cycles. The first option is not possible because the router needs the whole network packet to determine if it is ready but the core sends only the control signals and then the data signals if the router is ready. (The NoC router uses the AMBA protocol [41] for the ready and valid signals.) Therefore, the interface resets the ready signal to the core for 4 cycles after each request to stop the core to send any new requests until the router is ready.

Figure 5 show the changes in the tile after integrating the 2D-mesh NoC. After these changes, the memory accesses with local addresses are forwarded to the local memory whereas the rest of the accesses are forwarded to the network.

Figure 5: The structure of a generated tile before (left) and after (right) integrating the 2D mesh NoC.

In the third step, we set the number of rows and number of columns for the network, in the rocket chip generator, and generate the routers. During the fourth step, we cut the connections between the core and the memory module and rewire them to go through the network interface. Finally, based on the number of rows, columns and core ids, we connect the routers to each other to form the 2 dimensional mesh.

After the integration of the 2D-mesh NoC, generating manycore architectures can be performed by setting the configurations in the generator. It is
possible to generate a matrix of cores with arbitrary dimensions. The structure of the cores does not have to be a full matrix however due to the XY routing protocol the routers are generated in a complete mesh structure.

The crossbar network is partially removed from the generator, however a part of it still remains as the instructions are moved to the instruction cache through this network. The components such as the boot-ROM and the debug-ROM are on the shared address space. In order to access these components we keep the crossbar network between a core (core0) and these components. All other cores access these components through core0. In the future, those components might be connected to the 2D mesh network with proper interfaces.

2.5. Future Automation Directions

Being a first proof of concept, the design process still has a few manual tasks, which are necessary to generate an architecture and execute applications. These tasks are as follows:

- The instantiation and configuration of the cores within a new Chisel class that extends the configuration class within the rocket chip generator. The configurations include memory types and sizes, availability of FPU and accelerators, and many other details regarding the caches, memory lanes and other components.

- Configuration of the NoC interface, including setting the row and column sizes of the mesh network and the scratchpad memory size. By default this size is assumed to be the same for every core in the mesh. However, this can be configured.

In order to execute an application in a bare metal fashion on the generated emulators, the following steps are required:

- Development of new linker description for each different architecture. The code generation uses section attribute with the section names for each core. Hence, these names should be defined in the new linker description.

- Development of new start-up code if the number of the cores and the memory sizes change.
The configuration classes, linker description, and the start-up code can be generated automatically with an extension to the Cal2Many framework whereas for the NoC interface configuration, one may need to edit the interface to retrieve the configuration data from the configuration class. Nonetheless, the productivity of the developers will increase further with the automation of these tasks.

3. Case Studies

We have implemented two case studies to evaluate the design method, the software tools and the generated architectures. The case studies are a proof of concept to show that the design method works for different application domains. The first case study is different parallel versions of the autofocus criterion calculation [16]. The first version runs on a single core, the second version runs on 2 cores and the third version runs on 13 cores. The second case study is the first convolution layer of GoogLeNet [18] on 1, 4 and 5 cores. The main goal of this case study is not to achieve world leading performance, but to prove that the method can be used to improve the performance of the base, general purpose core. The applications are developed in CAL actor language and analyzed with TURNUS.

In the rest of this section, we will first provide the details of the implementations, followed by the practical details, to be able to generate the architectures and execute the implementations on them.

3.1. Autofocus Criterion Calculation

This application is a part of synthetic aperture radar systems [42]. These systems are usually mounted on flying platforms, where they use the motion of the platform over a target region to create two or 3-dimensional images of underlying objects. The path of the movement is not perfectly linear, however, additional processing can be performed to compensate this. The typical information used for the compensation is the positioning information from GPS [16]. However, in some cases this data might be insufficient. In such cases, the autofocus criterion can be used. There are different methods for calculating this criterion [43, 44]. The method that is used in this study, tries to find the flight path compensation that results in the best possible match between two images of the contributing subapertures. This requires several flight path compensations to be tested. The matching is checked with
a selected focus criterion. The criterion calculations, including interpolations and correlations, are performed many times.

The implementations consist of three main actions performing range interpolation (ranger), beam interpolation (beamer), and correlation (correlator) on 6x6 pixel kernels. Pixel information consists of position and color value represented as complex numbers. The real and imaginary parts of these numbers are represented with floating-point numbers. The ranger performs 36 cubic interpolations on each kernel, whereas the beamer performs 18 interpolations. The cubic interpolations are based on Neville’s algorithm [45]. Each interpolation take four input pixels and produces one pixel value in complex number format.

The ranger action takes a 6x6 kernel from one of the input images and applies interpolations to the rows. For each interpolation 4 pixels are consumed and a single pixel is produced. Hence the output of this action is a 6x3 matrix, which is forwarded to the beamer action as illustrated in Figure 6. Later, the same ranger action takes a kernel from the other input image and performs the same operations. The beamer action applies interpolation to the columns of the input. The result of the beam interpolation is a 3x3 matrix. Finally, the correlator action receives two consecutive 3x3 interpolation results (one for each image) from the beamer and calculates their correlation. After one iteration of computations, a single value is produced for each 2 input kernels. The flow of the data through the actions is illustrated in Figure 6. The two kernels, consisting of 36 pixel each, are reduced to a single floating-point number.

Figure 6: The dataflow implementation of the autofocus criterion calculation.
3.1.1. Application Development

The single core version is implemented as a single actor with the main actions (*ranger*, *beamer*, *correlator*) and several other helper actions for memory movement and scheduling. The actions are scheduled to run in a loop until all the input kernels are processed. There is no communication involved and the input data is stored within the actor. Figure 7 illustrates the structure of the actor.

![Figure 7: The single core implementation of the autofocus criterion calculation.](image)

The dual core version is implemented as 2 actors, where the first actor performs the range interpolation and the second actor performs the beam interpolation and correlation as seen in Figure 8. The results of the range interpolation are continuously sent to *actor2* through the communication channels where the beam interpolation and correlation are applied to the input data. The input data is stored in the first actor as seen in the figure.

![Figure 8: The dual core implementation of the autofocus criterion calculation.](image)

In the last version, we instantiated the structure of the dual core implementation 6 times and let the actors run in parallel to increase the data
parallelism. The number is chosen based on the input size, which is 6 kernels per image. The application can be parallelized further, however, it is not the focus of this study.

The correlation is a small task and it would be a waste of resources to use a correlation accelerator in each actor that performs the beam interpolation. A single accelerator is enough to perform the correlation efficiently for the entire application. However, the rocket chip generator does not support shared accelerators between cores. Therefore, we implemented the correlation operation as a separate actor and forward the results from the beam interpolators to this actor. This increases the task parallelism by letting the cores perform the correlation and the beam interpolation tasks in parallel. However, it adds a slight communication overhead. In total 6 actors are instantiated to perform the range interpolations, 6 actors are instantiated to perform the beam interpolations and one actor is instantiated to perform the correlation and sum all the results. Figure 9 shows the structure of this implementation. The input data is distributed to the actors, which perform range interpolations.

3.1.2. Analysis and Code Generation

In our previous study [9], we have already compared the results of single core architectures with and without accelerators. Therefore, in this study we do not generate any accelerators but only C code for the single core implementation. We execute the entire application on a single rocket core to get a reference point for performance comparisons.

The analysis results for the dual core implementation show the same results as the single core implementation and identify the cubic interpolations as the hot-spots of the actors. Therefore, the action that performs the interpolations is marked with the prefix \( \_\text{acc}\_ \) and an accelerator is generated for it. The only other part of the application where floating-point operations are performed is in the correlator. In order to avoid the floating-point unit (FPU) and save hardware resources, we marked the correlation action as a hot-spot and generated a second, smaller accelerator for the second actor to perform the correlation computations. The code generation results in parallel C code, to be compiled by the RISC-V gcc compiler and executed on two cores, and two different accelerators to be integrated to the these cores. The generated C code for the second core uses two different custom instructions to communicate with the accelerators.

We apply the same modifications to the correlation action in the imple-
Figure 9: The 13 core implementation of the autofocus criterion calculation.

mentation with 13 cores. The analysis and code generation processes are identical to the dual core implementation. Each actor is mapped onto an individual core and necessary communication mechanism is generated together with the two accelerators.

3.1.3. Accelerator Integration

The single core implementation does not have accelerators. Thus, we do not perform the accelerator integration step for this implementation. However, for the dual core implementation two different accelerators are generated. The first core executes the interpolation and therefore requires only the cubic interpolation accelerator, whereas the second core executes both the
interpolation and the correlation. Hence, it requires both the cubic interpolation and the correlation accelerators. The custom instructions are bound to the corresponding accelerators inside the configurations of the rocket chip generator. The RoCC interface is extended to instantiate both accelerators and establish the proper connections between the cores and the accelerators. Both accelerators require more than two 64 bits inputs. Therefore, the custom instructions are called more than once to transfer the inputs. The inputs are stored in the extended interface until the last instruction call that fires the accelerator arrives. We use a bit field in the custom instruction to distinguish between the calls. The accelerators for dual core and 13 core implementations are the same (cubic interpolation and correlation). Since the accelerators are already integrated to the rocket chip generator during the dual core implementation, there is no need to take this step for the 13 core implementation.

3.1.4. System Integration

The single core implementation does not require any system integration. However, in order to generate a new architecture with the rocket chip generator, one needs to add a Chisel class to the configurations of this tool, where the cores are instantiated and configured. The class for the single core implementation instantiates a tile consisting of a single core with an FPU, 4 KiB instruction cache and 256 KiB scratchpad memory.

The dual core implementation requires two tiles to be connected with the NoC. With the extensions described in this paper, the rocket chip generator generates the necessary 2D mesh network structure and connects the cores to the NoC routers. The network for two tiles is generated as a 1x2 matrix. The first core that computes the range interpolation is instantiated with a single accelerator. The other core computes both interpolation and correlation. Hence, it is instantiated with two accelerators. All of the floating-point operations computed in the application are executed on the accelerators. Therefore, no FPUs are used in the generated architectures. The rest of the configurations are kept identical to the single core implementation.

Due to the nature of XY routing we need to initiate a 4x4 mesh network with 16 routers for the 13 core implementation, even if only 13 of these routers are connected to a core. This architecture is illustrated in Figure 10. In the XY routing protocol, the packets travel first on the X axis and then on the Y axis. If the core in the last row sends a packet that is addressed to a core on the east then the packets will go through the routers, which are
not connected to any core. If the routers were not there the packets would be discarded.

The tiles in Figure 10 consist of a rocket core, 4 KiB instruction cache, 128 KiB scratchpad memory, an accelerator, and a NoC interface. When compared to the other implementations of this case study, the memory size is decreased in order to decrease the length of the critical path on the FPGA implementation. However, this does not affect the performance results. The tiles are identical except the one in the last row (Figure 10). The 12 tiles on the first three rows have the interpolation accelerator, whereas the tile on the fourth row has the correlation accelerator. The range interpolation actors in Figure 9 are mapped onto the first 6 cores when counted row-wise. The beam interpolation actors are mapped onto the second set of 6 cores and the correlation actor is mapped onto the core in the last row.

![Figure 10: The architecture generated for the 13 core implementation of the autofocus criterion calculation.](image)

### 3.2. The Convolution

In this case study we compute the first convolution layer of the GoogLeNet [18]. It is used as a proof of concept to show that the design method can be used to generate specialized architectures for machine learning domain.

The convolution size is $7 \times 7$ and the input image size is $224 \times 224$ in the RGB color space (padded to $230 \times 230$). The number of filters is 64 and the stride is 2. The image and the filters are stored as $3 \times 230 \times 230$ and $64 \times 7 \times 7$. 

---

27
matrices, respectively. With this configurations, the number of multiply and accumulate (mac) operations is 118,013,952.

3.2.1. Application Development

The convolution is implemented with three different approaches. The first approach is as a single actor with two actions namely applyFilter and accumulate. The applyFilter action performs the convolution using a 7x7 filter and a 7x7 area from one of the color dimensions of the image. The accumulate action stores the result of the applyFilter action and calculates the indices of the filter and the image matrices for the next convolution. At each iteration, the applyFilter calculates a result for one color dimension of the image. Thus, at every third iteration, the accumulate action accumulates three results of the applyFilter to produce an output pixel. In this first approach, the input image is stored in the actor.

In the second approach we use 5 actors to implement the convolution. One actor stores the input image and distributes it to three other actors. Each of these actors receives an area of 7x7 from only one of the R, G and B images, perform the convolution and send the results to a final actor that accumulates the R, G and B results. The actors that perform the convolution use the applyFilter action whereas the last actor uses the accumulate action.

In the last approach we store the R, G and B images directly in the actors, which compute the convolution to avoid the distribution of the image and consequently a large amount of on-chip communication. Therefore, we skip the actor that distributes the image and use only 4 actors; 3 actors to perform the convolution and 1 actor to accumulate the results from the convolution actors.

3.2.2. Analysis and Code Generation

The CAL implementations are fed to the TURNUS tool for the analysis. Predictably, the action that computes the convolution (applyFilter) is identified as the compute intensive part of the application. The analysis results of the single actor approach are presented in Table 1. The weights that are given in the table are products of operation weights, number of executed operations, and number of the firings. It is obvious that the applyFilter action is the compute intensive part of the actor with 98% of all the weight.

We added a hardware block, that performs convolution on two 7x7 input matrices, to our hardware library. In order to instantiate this hardware
Table 1: Bottleneck analysis of the actor performing the first convolution layer of the GoogLeNet (provided by TURNUS).

<table>
<thead>
<tr>
<th>Actor</th>
<th>Actions</th>
<th>Firings</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>applyFilter</td>
<td>2,408,448 50.0%</td>
<td>3,246,587,904.00 98.01%</td>
</tr>
<tr>
<td>conv</td>
<td>accumulate</td>
<td>2,408,448 50.0%</td>
<td>65,881,665.00 1.99%</td>
</tr>
</tbody>
</table>

block, we add a soft pragma to the application code. When the code generator goes through the action that will be converted into hardware, and finds a function call to a function (procedure in CAL) named `conv7`, it instantiates the hardware block and arranges the necessary connections. Therefore, we put the mac operations into the `conv7` function and call it within the `applyFilter` action. Figure 11 shows how the `conv7` pragma is added to the code. The computation within the action are replaced with a procedure call. The necessary calculations can be moved to the procedure. However, the content of the procedure is ignored during the code generation and the pre-defined hardware block is generated. The `conv7` procedure requires the filter index (to choose the right filter) and the first element of the image matrix as parameters. These parameters are then used as inputs to the accelerator and forwarded via custom instruction calls. The 64 filters used for the convolution are stored in the hardware block. This block requires the filter index to choose the right filter and the 7x7 input image to start the computations. Finally, the `applyFilter` action is converted into a hardware accelerator that instantiates the hardware block and the rest of the application is converted into C with custom instruction calls.

Using a library of manually implemented hardware blocks increases the efficiency of the hardware and enables re-use of the efficient hardware blocks, while decreasing the complexity and time consumption of the code generation.

While converting the multi-actor implementations, the same accelerator is generated for each actor that computes the convolution. For on-chip communication, the communication mechanism, that we integrated to our code generation tools, is used.

3.2.3. Accelerator Integration

The difference between the accelerator integration steps of the case studies is the usage of the memory connections. The accelerators of the first case study do not require the memory accesses due to receiving the input data
through the source registers of the custom instructions. However, in this case study, the input data to the accelerator is an index value for the filters and a 7x7 image that is relatively large. Therefore, instead of sending the whole image data, the core sends the address of the image data to the accelerator through the custom instruction. The interface, that is implemented manually by extending the RoCC interface, accesses the memory for the image data and forwards the memory response to the accelerator. The accelerator is fired when all the image data is sent to it. The result from the accelerator is forwarded directly to the core.

The accelerators, which are generated for each approach, are the same. Therefore we perform the accelerator integration once and use the same extended RoCC interface for each core that utilizes an accelerator.

3.2.4. System Integration

A single tile is generated for the single actor approach. The tile consists of core, memories and the accelerator. The scratchpad memory size of this tile is increased to 1 MiB due to the memory requirements. The core retains an FPU due to having floating-point operations outside the accelerated action (in the accumulate action). These operations can be moved to a small accelerator to avoid FPU and save hardware resources in a future work.

The 5 actor approach results in a 5 tile (or core) architecture. The first tile consists of 1 MiB memory as the image is stored in this tile. It acts as a storage. The other tiles have smaller memories (64 KiB) and only the tile that is generated for the accumulator actor has an FPU. Each tile that is
generated for the convolution actors has an accelerator.

The last generated architecture consists of 4 tiles. Since the input image is distributed to the tiles where the convolution is performed, each of them has a 256KiB memory. These tiles do not have FPU. The tile that accumulates the results have an FPU but a very small memory (16KiB).

4. Results and Discussions

The results that are considered for the evaluation of the design tools and the generated architectures are the performance (execution time), timing (max clock rate), and area (hardware resource usage). The configuration for the core in the rocket chip generator is based on the tiny configuration. We executed the case studies on the cycle accurate emulators for performance results. The timing and area results are provided by Xilinx synthesis tools. The generated verilog implementations are synthesized on a Xilinx VCU108 evaluation kit that features a Virtex Ultrascale XCVU095 FPGA.

4.1. Case Study 1 - Autofocus Criterion Calculation

We used the parallel implementations that we developed in this study together with the sequential implementation from a prior study [9] of the autofocus criterion calculation to evaluate the generation of single core, dual core and manycore architectures. The single core architecture is generated without an accelerator. Two different dual core architectures are generated for the dual core implementation. One of the architectures feature an FPU, whereas the other one use accelerators. The 13 core implementation is executed on cores without FPU and with accelerators. The instruction cache size for all cores is 4 KiB, whereas the scratchpad memory size is 256 KiB for single and dual core architectures and 128 KiB for the 13 core architecture. The implementations are tested with images, each consisting of 6 kernels. A kernel is a 6x6 pixel matrix. Half of the kernels overlap with each neighbour kernel both on row and column directions. Hence the input image size is 12x9.

Table 2 presents the performance results, whereas Table 3 presents the hardware resource usage and timing results for all versions of the implemented architectures. The transition from a single core architecture to a dual core architecture shows the effect of parallelization that increases the performance by a factor of 1.48 in terms of clock cycles. The resource usage doubles for memory and DSP units, however, some components of the
architecture are instantiated only once regardless the core count. Therefore, the usage of LUTs and FFs do not increase exactly by a factor of 2. The clock frequency decreases due to the size of the architecture. The hardware resources such as block RAMs (BRAM) and DSPs are distributed on the fabric. When more of these are needed, the resources from far corners of the fabric might require to be connected. Therefore, the wire delays become longer and contribute to the critical path. This situation re-occurs when the architecture size increase to 13 cores.

Table 2: Performance results of the generated architectures executing different implementations of the autofocus criterion calculation on cycle accurate emulators.

<table>
<thead>
<tr>
<th>Cycle count</th>
<th>Freq (MHz)</th>
<th>Time (ms)</th>
<th>Speed-up (cycles)</th>
<th>Speed-up (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core + FPU</td>
<td>830k</td>
<td>128</td>
<td>6.47</td>
<td>1</td>
</tr>
<tr>
<td>Dual core + FPU</td>
<td>560k</td>
<td>116</td>
<td>4.82</td>
<td>1.48×</td>
</tr>
<tr>
<td>Dual core + Acc</td>
<td>180k</td>
<td>116</td>
<td>1.55</td>
<td>4.6×</td>
</tr>
<tr>
<td>13 cores + Acc</td>
<td>31k</td>
<td>96</td>
<td>0.32</td>
<td>26.7×</td>
</tr>
</tbody>
</table>

Table 3: Hardware resource usage and timing results of the architectures generated for the autofocus criterion calculation on a Virtex Ultrascale XCVU095 FPGA.

<table>
<thead>
<tr>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Clock (MHz)</th>
<th>Memory (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core + FPU</td>
<td>27,353</td>
<td>9,039</td>
<td>65.5</td>
<td>15</td>
<td>128</td>
</tr>
<tr>
<td>Dual core + FPU</td>
<td>49,779</td>
<td>17,892</td>
<td>131</td>
<td>30</td>
<td>116</td>
</tr>
<tr>
<td>Dual core + Acc</td>
<td>33,570</td>
<td>16,432</td>
<td>133.5</td>
<td>91</td>
<td>116</td>
</tr>
<tr>
<td>13 cores + Acc</td>
<td>180,083</td>
<td>91,464</td>
<td>443</td>
<td>475</td>
<td>96</td>
</tr>
<tr>
<td>Total Available</td>
<td>537,600</td>
<td>1,075,200</td>
<td>1728</td>
<td>768</td>
<td>-</td>
</tr>
</tbody>
</table>

The difference between the dual core architecture results shows the effect of replacing FPUs with accelerators. This process increases the performance by 3.1×, while decreasing the LUT and FF usage. The BRAM utilization increases by a small amount while the DSP count increases significantly, which is an expected result of having accelerators.
When the number of cores is increased to 13 and each core is equipped with an accelerator, the performance (in terms of clock cycles) increases by a factor of 26.7 in comparison to the single core architecture. When compared to the dual core with accelerators, despite the increase of 6.5× in the core count, the performance increases by 5.8×. There are two main reasons behind not reaching the theoretical speed-up. The first one is the overhead of the communication between the beamer and correlator actors. The second one is the serialization process within the correlator. This actor reads the results of a beamer and performs correlation before reading the results of the next beamer. Since the chain of the ranger and the beamer actors are identical, their computation times are equal. Thus, they produce and send the outputs to the correlator actor at the same time. This creates a small queue and additional waiting time that contributes to the execution time. There can be different trade-off cases with the architecture of this application. One could keep the correlation within the beamer actors to achieve 6.5× speed-up, however, that would cost additional hardware resources for the correlation accelerator. Another approach to increase the performance could be using 6 cores instead of 1 core for the correlation. However, this would also increase the hardware resource usage, even more than the previous approach.

In case of resource usage, the increase is less than 6.5×. This is again due to some of the components being instantiated once, regardless the core count. Additionally, in the architecture with two cores, one of the cores has two accelerators, whereas in the 13 core architecture each core has a single accelerator. The number of interpolation accelerator increases to 12, however, the number of correlation accelerator does not change in the transition to 13 cores. It is clear that the number of accelerators does not increase by 6.5×.

To summarize, parallelization increases the performance as well as the hardware resource usage. Specialization increases the performance as well, however, in certain cases it might decrease the hardware resource usage as seen in the dual core implementations.

4.1.1. Comparison to Prior Studies

The same application is implemented and executed on different platforms in prior studies conducted by the members of our group [16, 17]. These platforms are Epiphany E16G3 and Intel i7-M260 as seen in Table 4. The table presents 3 different prior implementations from [16] and [17], and 2 different recent implementations from this study. The Intel results are given as a reference point. In terms of throughput, Intel and Epiphany platforms
outperform our platforms. However, our clock frequency results are from FPGA implementations. If implemented as ASICs, the clock frequencies should increase dramatically as the major parts of the critical paths are wires. For instance, if our 13 core architecture would run at 1GHz, the throughput/second would be 195,312 pixels, which is better than the 13 core Epiphany results by a small margin.

Single core of Epiphany outperforms our single core platform in cycles/pixels. However, when we replace the FPUs with accelerators, our platform shows a slightly better performance than the Epiphany architecture. These results are seen in Table 4, on the rows with 13 cores. The parallel implementation of the application, which run on the Epiphany is slightly more optimized than the implementation executed on our 13 core platform. The performance of our platform can be increased further by optimizing the software implementations.

Table 4: The performance comparison of the implementations of the autofocus criterion calculation on different platforms.

<table>
<thead>
<tr>
<th>Study</th>
<th>Platform</th>
<th>Number of cores</th>
<th>Throughput pixels/sec</th>
<th>Speed-up</th>
<th>Clock cycles/pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ul-Abdin et al. [17]</td>
<td>Intel i7, 2.67 GHz</td>
<td>1</td>
<td>21,600</td>
<td>1</td>
<td>123,611</td>
</tr>
<tr>
<td>Ul-Abdin et al. [16]</td>
<td>Epiphany, 1 GHz</td>
<td>1</td>
<td>17,668</td>
<td>0.8</td>
<td>56,600</td>
</tr>
<tr>
<td>This study</td>
<td>Rocket + FPU, 128 MHz</td>
<td>1</td>
<td>926</td>
<td>0.043</td>
<td>138,228</td>
</tr>
<tr>
<td>Ul-Abdin et al. [16]</td>
<td>Epiphany, 1 GHz</td>
<td>13</td>
<td>192,857</td>
<td>8.9</td>
<td>5185</td>
</tr>
<tr>
<td>This Study</td>
<td>Rocket + Acc, 96 MHz</td>
<td>13</td>
<td>18,750</td>
<td>0.87</td>
<td>5120</td>
</tr>
</tbody>
</table>

4.2. Case Study 2 - Convolution

We tested the convolution implementations on four different architectures. The first architecture is a single core with FPU, the second architecture is
a single core with FPU and accelerator, the third architecture has 5 cores, and the fourth architecture has 4 cores with different configurations. The first architecture is used as a reference point for comparison. All implementations perform 32 bit floating point calculations and therefore, do not aim to compete with other CNN implementations which use reduced precision.

Table 5 provide the performance results of these architectures where one can see that the performance of the single core architecture, while executing the application, is increased by a factor of 4, when the accelerator is integrated and utilized. Convolution of an image area of 7x7 (49 mac operations) takes approximately 800 cycles for the core (ignoring any memory overhead). The same operation takes 98 cycles for the accelerator including the 75 cycles taken for 49 memory accesses for the input image. In terms of computation, the accelerator is 8 times faster than the core. However, the application does not consist of only computations. The control operations such as scheduling and function calls (action in CAL) together with the accumulate function contribute to the execution time. Additionally, the core needs to perform address calculation to be able to send the address of the sub-image to the accelerator. The address calculation costs 4 loads, 4 multiplications and 3 additions and takes 23 cycles for the three dimensional input image. This contributes around 55M cycles to the execution time out of a total of 520M as seen in Table 5. In the future, with changes in the code generation tool, the address calculation can be moved to the accelerator and be executed faster.

The weight, that is generated through the analysis as an abstraction of the execution time, is 98% for the applyFilter action according to TURNUS. (This should not be confused with the weights used in convolutional neural networks (CNNs).) The analysis does not take the scheduler into account. However, when the CAL code is converted into C, the actions become functions and the scheduler becomes a state machine with if statements and calls to these functions. The scheduler adds an overhead to the execution time. Finally, with all the overheads, the applyFilter action takes around 86% of the execution time in the C implementation. Despite running the action 8 times faster with an accelerator, the overall speed up for the whole application becomes 4×.

In the 5-core architecture, three of the cores are utilized for the actual convolution computation. The impact of the accumulator core is not significant in this case. Therefore, the actual computation power is tripled and thus, the expectation is a 12× performance improvement when compared
to the reference architecture. However, the distribution of the input image adds a communication overhead that reduces the performance. Therefore the speed-up stays slightly below $12 \times$ (in terms of clock cycle).

The 4-core implementation is developed mainly to remove the communication overhead. Additionally, by removing the core that acts as a storage, we save hardware resources and the clock rate becomes higher. The computation power is the same as the 5-core implementation, however, the speed-up is higher than $12 \times$. This is mainly due to performing fewer operations for the address calculation while sending the sub-image address to the accelerator. Additionally, the schedulers are smaller due to dividing the convolution and the accumulation functions onto different cores.

In the 4-core implementation, the input image is divided into R, G and B images and stored on separate cores as two dimensional arrays. Calculating an address within these arrays require two multiplications and two additions. This is two multiplications and one addition fewer than the single core implementation.

Table 5: Performance results of the architectures generated for the convolution task.

<table>
<thead>
<tr>
<th></th>
<th>Cycle count</th>
<th>Freq (MHz)</th>
<th>Time (second)</th>
<th>Speed-up cycles</th>
<th>Speed-up seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>2,121,880,320</td>
<td>113</td>
<td>18.7</td>
<td>1×</td>
<td>1×</td>
</tr>
<tr>
<td>+ FPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single core</td>
<td>520,475,648</td>
<td>113</td>
<td>4.6</td>
<td>4×</td>
<td>4 ×</td>
</tr>
<tr>
<td>+ FPU + Acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 cores + 3 Acc + 1 FPU</td>
<td>200,660,096</td>
<td>100</td>
<td>2.0</td>
<td>10.5×</td>
<td>9.35 ×</td>
</tr>
<tr>
<td>4 cores + 3 Acc + 1 FPU</td>
<td>145,480,775</td>
<td>111</td>
<td>1.3</td>
<td>14.5×</td>
<td>14.3×</td>
</tr>
</tbody>
</table>

The clock frequency differs between the single core architectures generated for different case studies. The architecture that is generated for the first case study runs at 128 MHz, whereas the architecture for the second study runs at 113 MHz. The reason of this difference is the utilization of the block RAMs (BRAMs). The first architecture has 256 KiB scratchpad memory and require 66 BRAMs, while the second architecture require 258 BRAMs due to
having 1 MiB scratchpad memory. Since the BRAMs are distributed on the FPGA fabric, utilizing more of them require longer connections. This results in longer critical path and consequently lower clock frequencies. The details of the resource usage results are given in Table 6. The accelerator is larger than the previous accelerators. It features 49 (32-bit) floating-point multipliers, 48 (32-bit) floating-point adders, and 64 filters. Each filter consists of $7 \times 7$ 32-bit floating point numbers. The DSPs are used by the multipliers. The convolution hardware block that is instantiated within the generated accelerator is developed and added to our hardware library within a few days. With the design tools, it can be instantiated instantly.

The distributor core in the 5-core architecture has 1 MiB scratchpad memory and no FPU. The three computation cores have 64 KiB memory and an accelerator each. The accumulator core has 64 KiB memory and an FPU. The memory of the cores can be decreased to reduce the BRAM utilization. However, the storage/distributor core has to have around 634,800 bytes just for the input image. The LUT and FF usage of the 5-core architecture is respectively 4 and 6 times higher than the usage of the reference architecture. The number of the block RAMs is increased by 70 whereas the DSP usage is increased by factor of 21. These increases are irregular due to the heterogeneous nature of the 5 core architecture. The 4-core architecture has a similar behaviour in terms of hardware resource usage. In order to optimize this usage and increase the clock rate, we removed the distributor core. We used 256 KiB of memory in each computation core, which sums up to 768 KiB in total (for storing the entire image together with the other data and the code). Additionally, we reduced the memory size of the accumulator core as well. Therefore, the BRAM utilization is decreased from 327.5 to 202. The decrease in the resource usage allows a higher clock rate.

The CNN applications usually use lower precision to represent the numbers. In the future, the implementations and the hardware block for the convolution can be optimized further to use fewer bits to represent the numbers. This will lead to smaller accelerators, less memory usage, higher clock rate and higher performance.

5. Conclusions

The performance requirements of today’s applications pushes the computer architectures towards parallelism and specialization. However, designing or exploring the design space of specialized architectures with multiple
cores is a substantial challenge. In this paper we address this challenge and propose a design method that can generate application specific or domain specific manycore architectures with software tools that automate the steps of the method. The tools used in the method allows generation of application specific accelerators directly from the application code written in a dataflow language as well as the manycore architectures with different configurations including number of cores, memory types & sizes, and availability of certain components (FPU, accelerator, etc.).

The proposed design method facilitates the development of specialized (heterogeneous) manycore architectures with different configurations. It decreases the architecture design work of days and months into minutes or hours. This allows the developers to test their applications on different architectures or generate a specific architecture for the application in hand. The results show that the specialized architectures demonstrate a better performance than general purpose architectures. They might even reduce the area usage of the architecture in certain cases by removing large, general purpose components such as FPU. Otherwise, the area of the architectures increase when they are equipped with application specific accelerators. The clock frequency of FPGA implementations currently decrease when the number of cores increase due to the placement of the memory resources. However, with an ASIC implementation where the hardware resources can be placed arbitrarily, the clock frequency should not decrease when the architectures scale-up due to having encapsulated tiles.
We believe that, computer architectures will continue to move towards parallelism and heterogeneity. Therefore, a fully automated, open design method with a large library of hardware blocks performing different computations will be an essential tool for exploring the design space for new architectures.

References


43